COMPSCI 220S2C, 2007

Mark C. Wilson

August 9, 2007

1. Introduction and Background
   - Heapsort

2. Analysis of Searching
   - List implementations
   - Tree implementations
   - Hashing implementations

Selection sort improvement idea

- In selection sort, finding the minimum of $a[i..n-1]$ by sequential search is slow, and it dominates the running time. Can we do this operation faster?

Selection sort improvement idea

- In selection sort, finding the minimum of $a[i..n-1]$ by sequential search is slow, and it dominates the running time. Can we do this operation faster?
- Not with the current data structure. What about a different one?

Selection sort improvement idea

- In selection sort, finding the minimum of $a[i..n-1]$ by sequential search is slow, and it dominates the running time. Can we do this operation faster?
- Not with the current data structure. What about a different one?
- We want a data structure that allows us to find and extract the minimum quickly.

Priority queues

- Recall that a priority queue is a container ADT where each element has a key called its priority. There are operations allowing us to insert an element, and to find and delete the element of highest priority.

Priority queues

- Recall that a priority queue is a container ADT where each element has a key called its priority. There are operations allowing us to insert an element, and to find and delete the element of highest priority.
- Queues and stacks are special cases that are more efficiently done directly.

Priority queues

- Recall that a priority queue is a container ADT where each element has a key called its priority. There are operations allowing us to insert an element, and to find and delete the element of highest priority.
- Queues and stacks are special cases that are more efficiently done directly.
- Priority queues are important in many areas: discrete event simulation, graph algorithms (later in this course), sorting.

Priority queues

- Recall that a priority queue is a container ADT where each element has a key called its priority. There are operations allowing us to insert an element, and to find and delete the element of highest priority.
- Queues and stacks are special cases that are more efficiently done directly.
- Priority queues are important in many areas: discrete event simulation, graph algorithms (later in this course), sorting.
- Priority queues can be implemented in many ways: unsorted list, sorted list, binary heap, binomial heap, ...

Priority queue sort

- Suppose we are given an input list. Start with an empty priority queue Q and:

Priority queue sort

- Suppose we are given an input list. Start with an empty priority queue Q and:
  - successively insert all elements into Q, using the sorting keys as the priority;

Priority queue sort

- Suppose we are given an input list. Start with an empty priority queue Q and:
  - successively insert all elements into Q, using the sorting keys as the priority;
  - successively remove the highest priority element until Q is empty.

Priority queue sort

- Suppose we are given an input list. Start with an empty priority queue Q and:
  - successively insert all elements into Q, using the sorting keys as the priority;
  - successively remove the highest priority element until Q is empty.
- This gives a sorting algorithm that is obviously correct. Performance:

Priority queue sort

- Suppose we are given an input list. Start with an empty priority queue Q and:
  - successively insert all elements into Q, using the sorting keys as the priority;
  - successively remove the highest priority element until Q is empty.
- This gives a sorting algorithm that is obviously correct. Performance:
  - If we implement Q using an unsorted list, we obtain selection sort. Insertion takes $O(1)$ time but deletion time is $\Theta(n)$.

Priority queue sort

- Suppose we are given an input list. Start with an empty priority queue Q and:
  - successively insert all elements into Q, using the sorting keys as the priority;
  - successively remove the highest priority element until Q is empty.
- This gives a sorting algorithm that is obviously correct. Performance:
  - If we implement Q using an unsorted list, we obtain selection sort. Insertion takes $O(1)$ time but deletion time is $\Theta(n)$.
  - If we use a sorted list to implement Q, we obtain insertion sort. Insertion takes $\Theta(n)$ time but deletion is $O(1)$.

Priority queue sort

- Suppose we are given an input list. Start with an empty priority queue Q and:
  - successively insert all elements into Q, using the sorting keys as the priority;
  - successively remove the highest priority element until Q is empty.
- This gives a sorting algorithm that is obviously correct. Performance:
  - If we implement Q using an unsorted list, we obtain selection sort. Insertion takes $O(1)$ time but deletion time is $\Theta(n)$.
  - If we use a sorted list to implement Q, we obtain insertion sort. Insertion takes $\Theta(n)$ time but deletion is $O(1)$.
  - We can do better with an implementation in which insertion and deletion each take time in $O(\log n)$. The simplest is the binary heap.

Priority queue sort pseudocode

algorithm $\texttt{pqsort}$ (list a)
$Q \leftarrow \texttt{pqbuild}(a)$
$t = \texttt{list}()$
while not $Q.\texttt{empty}$
    t.add(**delete**(Q))
return t
end

Priority queue: binary heap implementation

- A heap is a binary tree that

Priority queue: binary heap implementation

- A heap is a binary tree that
  - is complete (every level except perhaps the last is full and the last level is left-filled)

Priority queue: binary heap implementation

- A heap is a binary tree that
  - is complete (every level except perhaps the last is full and the last level is left-filled)
  - has the partial order property (on every path from the root, the keys decrease).

Priority queue: binary heap implementation

- A heap is a binary tree that
  - is complete (every level except perhaps the last is full and the last level is left-filled)
  - has the partial order property (on every path from the root, the keys decrease).
- A heap can implement a priority queue as follows.

Priority queue: binary heap implementation

- A heap is a binary tree that
  - is complete (every level except perhaps the last is full and the last level is left-filled)
  - has the partial order property (on every path from the root, the keys decrease).
- A heap can implement a priority queue as follows.
  - Keys are stored in nodes.

Priority queue: binary heap implementation

- A heap is a binary tree that
  - is complete (every level except perhaps the last is full and the last level is left-filled)
  - has the partial order property (on every path from the root, the keys decrease).
- A heap can implement a priority queue as follows.
  - Keys are stored in nodes.
  - To insert a node, create a new leaf at the bottom level as far left as possible. Swap it upward until no swap is required.

Priority queue: binary heap implementation

- A heap is a binary tree that
  - is complete (every level except perhaps the last is full and the last level is left-filled)
  - has the partial order property (on every path from the root, the keys decrease).
- A heap can implement a priority queue as follows.
  - Keys are stored in nodes.
  - To insert a node, create a new leaf at the bottom level as far left as possible. Swap it upward until no swap is required.
  - To delete the maximum, remove the root. Put the rightmost leaf in the root position. Swap it downward (choosing the larger child each time) until no swap is required.

Priority queue: binary heap implementation

- A heap is a binary tree that
  - is complete (every level except perhaps the last is full and the last level is left-filled)
  - has the partial order property (on every path from the root, the keys decrease).
- A heap can implement a priority queue as follows.
  - Keys are stored in nodes.
  - To insert a node, create a new leaf at the bottom level as far left as possible. Swap it upward until no swap is required.
  - To delete the maximum, remove the root. Put the rightmost leaf in the root position. Swap it downward (choosing the larger child each time) until no swap is required.
- A heap is sometimes called a tournament.

Array representation of binary heap

- A binary heap can be represented implicitly (without pointers) in an array.

Array representation of binary heap

- A binary heap can be represented implicitly (without pointers) in an array.
    - Each node corresponds to an array index (starting with 1).

Array representation of binary heap

- A binary heap can be represented implicitly (without pointers) in an array.
  - Each node corresponds to an array index (starting with 1).
  - The children of the node corresponding to index k are in positions $2k$ (left) and $2k+1$ (right). The keys are stored directly in the array.

Array representation of binary heap

- A binary heap can be represented implicitly (without pointers) in an array.
  - Each node corresponds to an array index (starting with 1).
  - The children of the node corresponding to index k are in positions $2k$ (left) and $2k+1$ (right). The keys are stored directly in the array.
- To insert a key x, put it in position $n+1$ and swap as above.

Array representation of binary heap

- A binary heap can be represented implicitly (without pointers) in an array.
  - Each node corresponds to an array index (starting with 1).
  - The children of the node corresponding to index k are in positions $2k$ (left) and $2k+1$ (right). The keys are stored directly in the array.
- To insert a key x, put it in position $n+1$ and swap as above.
- To delete the root, swap $a[1]$ with $a[n]$, and swap as above. Note that deleted elements end up at the end of the array.

Heapsort analysis

- Building a heap using n successive insertions takes time in $O(n \log n)$ since the tree has height in $O(\log n)$.

Heapsort analysis

- Building a heap using n successive insertions takes time in $O(n \log n)$ since the tree has height in $O(\log n)$.
- Deleting the root n times, restoring the heap property each time, takes time in $O(n \log n)$.

Heapsort analysis

- Building a heap using n successive insertions takes time in $O(n \log n)$ since the tree has height in $O(\log n)$.
- Deleting the root n times, restoring the heap property each time, takes time in $O(n \log n)$.
- Thus heapsort is a worst-case $\Theta(n \log n)$ sorting algorithm, like mergesort. It is not stable, but it is in-place.

Heapsort analysis

- Building a heap using n successive insertions takes time in $O(n \log n)$ since the tree has height in $O(\log n)$.
- Deleting the root n times, restoring the heap property each time, takes time in $O(n \log n)$.
- Thus heapsort is a worst-case $\Theta(n \log n)$ sorting algorithm, like mergesort. It is not stable, but it is in-place.
- Detailed average-case analysis is more difficult, but the best and worst case are not very different.

Heapsort variants

- We can build the heap by repeatedly adding elements to the empty heap. However, for sorting we only need the heap property at the end.

Heapsort variants

- We can build the heap by repeatedly adding elements to the empty heap. However, for sorting we only need the heap property at the end.
- One way is to build a complete binary tree without the heap property. Then recursively heapify the left and right subtrees, and then let the root swap down to the right position.

Heapsort variants

- We can build the heap by repeatedly adding elements to the empty heap. However, for sorting we only need the heap property at the end.
- One way is to build a complete binary tree without the heap property. Then recursively heapify the left and right subtrees, and then let the root swap down to the right position.
- The recursion $T(n) = 2T(n/2) + \lg n$ describes the running time of the latter method: solution is $\Theta(n)$.

Heapsort variants

- We can build the heap by repeatedly adding elements to the empty heap. However, for sorting we only need the heap property at the end.
- One way is to build a complete binary tree without the heap property. Then recursively heapify the left and right subtrees, and then let the root swap down to the right position.
- The recursion $T(n) = 2T(n/2) + \lg n$ describes the running time of the latter method: solution is $\Theta(n)$.
- This method can be rewritten to avoid recursion ("bottom-up") and to work in place in the array implementation.

Heapsort variants

- We can build the heap by repeatedly adding elements to the empty heap. However, for sorting we only need the heap property at the end.
- One way is to build a complete binary tree without the heap property. Then recursively heapify the left and right subtrees, and then let the root swap down to the right position.
- The recursion $T(n) = 2T(n/2) + \lg n$ describes the running time of the latter method: solution is $\Theta(n)$.
- This method can be rewritten to avoid recursion ("bottom-up") and to work in place in the array implementation.
- There is also a way to delete the maximum that is about twice as fast on average: it involves swapping down a long way and then swapping back up if necessary. See me for more details.

Heapsort variants

- We can build the heap by repeatedly adding elements to the empty heap. However, for sorting we only need the heap property at the end.
- One way is to build a complete binary tree without the heap property. Then recursively heapify the left and right subtrees, and then let the root swap down to the right position.
- The recursion $T(n) = 2T(n/2) + \lg n$ describes the running time of the latter method: solution is $\Theta(n)$.
- This method can be rewritten to avoid recursion ("bottom-up") and to work in place in the array implementation.
- There is also a way to delete the maximum that is about twice as fast on average: it involves swapping down a long way and then swapping back up if necessary. See me for more details.
- There is still serious research being done on better priority queue implementations.

Summary of sorting algorithms

Table: Characteristics of sorting methods

Method	Worst	Average	Best	Stable?	In-place?
Insertion	n^2	n^2	n	Yes	Yes
Selection	n^2	n^2	n^2	No	Yes
Shellsort	??	??	n	No	Yes
Mergesort	$n \log n$	$n \log n$	$n \log n$	Yes	No
Quicksort	n^2	$n \log n$	$n \log n$	No	Almost
Heapsort	$n \log n$	$n \log n$	$n \log n$	No	Yes

Running times give asymptotic order only. Shellsort analysis depends on the increments used, and is difficult. Quicksort needs a stack of size $\Theta(\log n)$ for the recursion.

Selection

- Fix r with $1 \leq r \leq n$. We want to find the element with rth key from an input list (the rth order statistic). Should be easier than sorting!
- Building a priority queue and extracting is probably too much work even if $r = 1$, certainly if $r = n/2$.
- One approach is to use the quicksort idea. At each stage we only need to make a recursive call on one half of the array because we know where the pivot is relative to the desired element.
- The recurrence for the average number of comparisons has the form $E(n) = n + \frac{1}{n} \sum_{i=0}^{n-1} E(p)$. This has a solution that is $\Theta(n)$. See textbook for details.
- The worst case is still quadratic; there is another divide-and-conquer algorithm that is worst-case linear, but more complicated (see COMPSCI 320).

Extra: sorting analysis - where to from here?

- Basic calculations that we have performed give the worst and average case running time, which is enough to rule out certain algorithms as being competitive for large input.
- More precise analysis (such as covered in COMPSCI 720) allows us to choose parameters (such as quicksort cutoff, pivot selection strategies) to optimize performance, and to make finer comparisons between algorithms.
- The more detail is required, the more advanced the mathematical machinery needed. The mathematics involved in modern research makes heavy use of advanced calculus techniques even though it is about discrete quantities.
- See Flajolet and Sedgewick, *Introduction to the Analysis of Algorithms*; Knuth, *The Art of Computer Programming*; me, for a research project.

Table ADT

- Recall that a table is a container ADT which supports the operations of finding a key or reporting that it is not present, retrieving a record with given key, inserting a record, deleting a record.

Table ADT

- Recall that a table is a container ADT which supports the operations of finding a key or reporting that it is not present, retrieving a record with given key, inserting a record, deleting a record.
- The set of keys need not be totally ordered, but no keys can be repeated.

Table ADT

- Recall that a table is a container ADT which supports the operations of finding a key or reporting that it is not present, retrieving a record with given key, inserting a record, deleting a record.
- The set of keys need not be totally ordered, but no keys can be repeated.
- A table can perform the tasks done by a priority queue but is much more general.

Table ADT

- Recall that a table is a container ADT which supports the operations of finding a key or reporting that it is not present, retrieving a record with given key, inserting a record, deleting a record.
- The set of keys need not be totally ordered, but no keys can be repeated.
- A table can perform the tasks done by a priority queue but is much more general.
- There are many implementations: (sorted or unsorted) list, hash table and (various types of) binary search trees are the main ones. Also skip lists, jump lists.

Table ADT

- Recall that a table is a container ADT which supports the operations of finding a key or reporting that it is not present, retrieving a record with given key, inserting a record, deleting a record.
- The set of keys need not be totally ordered, but no keys can be repeated.
- A table can perform the tasks done by a priority queue but is much more general.
- There are many implementations: (sorted or unsorted) list, hash table and (various types of) binary search trees are the main ones. Also skip lists, jump lists.
- Static searching does not perform insertions or deletions (such as in a telephone book), while dynamic searching allows insertion and deletion (such as in a database).

Sequential search

- In a list with no other structure, the only way to find an element is to check each element.

Sequential search

- In a list with no other structure, the only way to find an element is to check each element.
- This takes time in $\Theta(n)$ for any reasonable implementation (such as array or linked list).

Sequential search

- In a list with no other structure, the only way to find an element is to check each element.
- This takes time in $\Theta(n)$ for any reasonable implementation (such as array or linked list).
- We only need to be able to iterate through the elements in linear time, so even more general structures than lists also allow for this type of search.

Binary search

- In a sorted list where constant time access is possible (such as an array implementation), we can find a key x as follows. Start at the middle key, and recursively go left or right if the key is greater or less than x; stop if we hit x or search subinterval is empty.

Binary search

- In a sorted list where constant time access is possible (such as an array implementation), we can find a key x as follows. Start at the middle key, and recursively go left or right if the key is greater or less than x; stop if we hit x or search subinterval is empty.
- Note: binary search is conceptually easy but surprisingly hard to program correctly even for professionals (see J. Bentley, *Programming Pearls*).

Binary search

- In a sorted list where constant time access is possible (such as an array implementation), we can find a key x as follows. Start at the middle key, and recursively go left or right if the key is greater or less than x; stop if we hit x or search subinterval is empty.
- Note: binary search is conceptually easy but surprisingly hard to program correctly even for professionals (see J. Bentley, *Programming Pearls*).
- The (worst-case) recurrence is $T(n) = 1 + T(\lfloor n/2 \rfloor)$, with solution in $O(\log n)$.

Binary search

- In a sorted list where constant time access is possible (such as an array implementation), we can find a key x as follows. Start at the middle key, and recursively go left or right if the key is greater or less than x; stop if we hit x or search subinterval is empty.
- Note: binary search is conceptually easy but surprisingly hard to program correctly even for professionals (see J. Bentley, *Programming Pearls*).
- The (worst-case) recurrence is $T(n) = 1 + T(\lfloor n/2 \rfloor)$, with solution in $O(\log n)$.
- The execution of this algorithm (looking for all possible keys) can be described by a decision tree called a (static) binary search tree. The number of comparisons required to find the key is the depth of the leaf containing that key.

Binary search trees

- A binary search tree is a binary tree with keys stored in nodes, such that the key of each node is $\geq$ the key of its left child and $\leq$ the key of its right child.

Binary search trees

- A binary search tree is a binary tree with keys stored in nodes, such that the key of each node is $\geq$ the key of its left child and $\leq$ the key of its right child.
- A BST implements the Table ADT. To find/insert a key, use binary search as described above.

Binary search trees

- A binary search tree is a binary tree with keys stored in nodes, such that the key of each node is $\geq$ the key of its left child and $\leq$ the key of its right child.
- A BST implements the Table ADT. To find/insert a key, use binary search as described above.
- To remove a node, we need more work.

Binary search trees

- A binary search tree is a binary tree with keys stored in nodes, such that the key of each node is $\geq$ the key of its left child and $\leq$ the key of its right child.
- A BST implements the Table ADT. To find/insert a key, use binary search as described above.
- To remove a node, we need more work.
    - An internal node with no children: simply delete.

Binary search trees

- A binary search tree is a binary tree with keys stored in nodes, such that the key of each node is $\geq$ the key of its left child and $\leq$ the key of its right child.
- A BST implements the Table ADT. To find/insert a key, use binary search as described above.
- To remove a node, we need more work.
  - An internal node with no children: simply delete.
  - An internal node with only one child: delete the node, connect the child to the parent.

Binary search trees

- A binary search tree is a binary tree with keys stored in nodes, such that the key of each node is $\geq$ the key of its left child and $\leq$ the key of its right child.
- A BST implements the Table ADT. To find/insert a key, use binary search as described above.
- To remove a node, we need more work.
  - An internal node with no children: simply delete.
  - An internal node with only one child: delete the node, connect the child to the parent.
  - An internal node n with two children: find the minimum key K in the right subtree, delete that node, and replace the key of n by K.

Binary search trees

- A binary search tree is a binary tree with keys stored in nodes, such that the key of each node is $\geq$ the key of its left child and $\leq$ the key of its right child.
- A BST implements the Table ADT. To find/insert a key, use binary search as described above.
- To remove a node, we need more work.
  - An internal node with no children: simply delete.
  - An internal node with only one child: delete the node, connect the child to the parent.
  - An internal node n with two children: find the minimum key K in the right subtree, delete that node, and replace the key of n by K.
- BSTs are very versatile. They are good for sorting. An inorder traversal of the tree yields the keys in sorted order. Also, BSTs model the behaviour of quicksort.

Binary search trees

- A binary search tree is a binary tree with keys stored in nodes, such that the key of each node is $\geq$ the key of its left child and $\leq$ the key of its right child.
- A BST implements the Table ADT. To find/insert a key, use binary search as described above.
- To remove a node, we need more work.
  - An internal node with no children: simply delete.
  - An internal node with only one child: delete the node, connect the child to the parent.
  - An internal node n with two children: find the minimum key K in the right subtree, delete that node, and replace the key of n by K.
- BSTs are very versatile. They are good for sorting. An inorder traversal of the tree yields the keys in sorted order. Also, BSTs model the behaviour of quicksort.
- Main problem with BSTs: they can become unbalanced by insertions and deletions.

Analysis of BST operations

- The running time of all basic operations is proportional to the number of nodes visited.

Analysis of BST operations

- The running time of all basic operations is proportional to the number of nodes visited.
- In the worst case, finding/removing/inserting take time in $\Theta(h)$ where h is the height of the tree. Unfortunately the height can be as large as $n - 1$ in the worst case.

Analysis of BST operations

- The running time of all basic operations is proportional to the number of nodes visited.
- In the worst case, finding/removing/inserting take time in $\Theta(h)$ where h is the height of the tree. Unfortunately the height can be as large as $n - 1$ in the worst case.
- A BST built by n insertions of random keys has height in $\Theta(\log n)$ (see next slide). So randomly grown trees are not too unbalanced.

Analysis of BST operations

- The running time of all basic operations is proportional to the number of nodes visited.
- In the worst case, finding/removing/inserting take time in $\Theta(h)$ where h is the height of the tree. Unfortunately the height can be as large as $n-1$ in the worst case.
- A BST built by n insertions of random keys has height in $\Theta(\log n)$ (see next slide). So randomly grown trees are not too unbalanced.
- However deletions can mess this up, and are hard to analyse.

Analysis of BST operations

- The running time of all basic operations is proportional to the number of nodes visited.
- In the worst case, finding/removing/inserting take time in $\Theta(h)$ where h is the height of the tree. Unfortunately the height can be as large as $n - 1$ in the worst case.
- A BST built by n insertions of random keys has height in $\Theta(\log n)$ (see next slide). So randomly grown trees are not too unbalanced.
- However deletions can mess this up, and are hard to analyse.
- Conclusion: we need another idea to guarantee good worst-case performance. We need to rebalance BSTs.

Relation between Quicksort and BSTs

- After choosing the pivot and partitioning, we can represent the file as a binary tree: pivot at the root, left subfile on the left, right subfile on the right. It has the BST property with respect to the sort keys.
- Given the above BST describing the execution of quicksort on the file, note that the cost of constructing the tree (measured by key comparisons) is the same as the number of comparisons used by quicksort in sorting the file.
- This is equal to the internal path length, the sum of all depths of nodes.
- Thus the average search cost is $\Theta(\log n)$ for randomly grown BSTs with no deletions.

Extra: self-balancing binary search trees

- There are several classes of BSTs (such as AVL, red-black, AA) that perform rebalancing operations at crucial times in order to keep the height close to $\lg n$.

Extra: self-balancing binary search trees

- There are several classes of BSTs (such as AVL, red-black, AA) that perform rebalancing operations at crucial times in order to keep the height close to $\lg n$.
- These operations are based on local rotations of the tree.

Extra: self-balancing binary search trees

- There are several classes of BSTs (such as AVL, red-black, AA) that perform rebalancing operations at crucial times in order to keep the height close to $\lg n$.
- These operations are based on local rotations of the tree.
- Analysis of performance is fairly difficult. It is not too hard to show that AVL trees maintain the right height (see textbook). Average-case height is not really known.

Extra: self-balancing binary search trees

- There are several classes of BSTs (such as AVL, red-black, AA) that perform rebalancing operations at crucial times in order to keep the height close to $\lg n$.
- These operations are based on local rotations of the tree.
- Analysis of performance is fairly difficult. It is not too hard to show that AVL trees maintain the right height (see textbook). Average-case height is not really known.
- Java Collection Framework's `TreeMap` uses red-black tree implementation.

Hashing

- A hash function is a function h that outputs an integer value for each key. A hash table is an array implementation of the table ADT, where each key is mapped via a hash function to an array index.

Hashing

- A hash function is a function h that outputs an integer value for each key. A hash table is an array implementation of the table ADT, where each key is mapped via a hash function to an array index.
- The number of possible keys is usually enormously more than the actual number of keys. Thus allocating an array with enough size to fit all possible keys would be very inefficient. So hash functions are not 1-to-1; that is, two keys may be mapped to the same index (a collision).

Hashing

- A hash function is a function h that outputs an integer value for each key. A hash table is an array implementation of the table ADT, where each key is mapped via a hash function to an array index.
- The number of possible keys is usually enormously more than the actual number of keys. Thus allocating an array with enough size to fit all possible keys would be very inefficient. So hash functions are not 1-to-1; that is, two keys may be mapped to the same index (a collision).
- We need a collision resolution policy to prescribe what to do when collisions occur.

Hashing

- A hash function is a function h that outputs an integer value for each key. A hash table is an array implementation of the table ADT, where each key is mapped via a hash function to an array index.
- The number of possible keys is usually enormously more than the actual number of keys. Thus allocating an array with enough size to fit all possible keys would be very inefficient. So hash functions are not 1-to-1; that is, two keys may be mapped to the same index (a collision).
- We need a collision resolution policy to prescribe what to do when collisions occur.
- We assume the first-come-first served (FCFS) model for resolving collisions.

Hash functions

- There are several desirable properties of a hash function:

Hash functions

- There are several desirable properties of a hash function:
  - it should be computable quickly (constant time).

Hash functions

- There are several desirable properties of a hash function:
  - it should be computable quickly (constant time).
  - if keys are drawn uniformly at random, then the hashed values should be uniformly distributed.

Hash functions

- There are several desirable properties of a hash function:
    - it should be computable quickly (constant time).
    - if keys are drawn uniformly at random, then the hashed values should be uniformly distributed.
    - keys that are “close” should have their hash values “spread out”.

Hash functions

- There are several desirable properties of a hash function:
  - it should be computable quickly (constant time).
  - if keys are drawn uniformly at random, then the hashed values should be uniformly distributed.
  - keys that are "close" should have their hash values "spread out".
- A hash function should be deterministic, but appear "random" - in other words it should pass some statistical tests (similar to pseudorandom number generators).

Hash functions

- There are several desirable properties of a hash function:
  - it should be computable quickly (constant time).
  - if keys are drawn uniformly at random, then the hashed values should be uniformly distributed.
  - keys that are “close” should have their hash values “spread out”.
- A hash function should be deterministic, but appear “random” - in other words it should pass some statistical tests (similar to pseudorandom number generators).
- Example: Java String `hashcode` computes the address of a string s using integer arithmetic via the formula $s[0] * 31^{n-1} + s[1] * 31^{n-2} + \ldots + s[n-1]$.

Collision resolution policies

- Open addressing uses no extra space - every element is stored in the hash table. If it gets overfull, we can reallocate space and rehash.

Collision resolution policies

- Open addressing uses no extra space - every element is stored in the hash table. If it gets overfull, we can reallocate space and rehash.
- Chaining uses an “overflow” list for each element in the hash table.

Collision resolution policies

- Open addressing uses no extra space - every element is stored in the hash table. If it gets overfull, we can reallocate space and rehash.
- Chaining uses an "overflow" list for each element in the hash table.
- When a collision occurs, we can evict the occupant (LCFS), evict the latecomer (FCFS) or evict the element furthest from its home location (Robin Hood hashing).

Collision resolution policies

- Open addressing uses no extra space - every element is stored in the hash table. If it gets overfull, we can reallocate space and rehash.
- Chaining uses an “overflow” list for each element in the hash table.
- When a collision occurs, we can evict the occupant (LCFS), evict the latecomer (FCFS) or evict the element furthest from its home location (Robin Hood hashing).
- When a collision occurs in open addressing, we can

Collision resolution policies

- Open addressing uses no extra space - every element is stored in the hash table. If it gets overfull, we can reallocate space and rehash.
- Chaining uses an "overflow" list for each element in the hash table.
- When a collision occurs, we can evict the occupant (LCFS), evict the latecomer (FCFS) or evict the element furthest from its home location (Robin Hood hashing).
- When a collision occurs in open addressing, we can
  - probe nearby for a free position (linear probing, quadratic probing);

Collision resolution policies

- Open addressing uses no extra space - every element is stored in the hash table. If it gets overfull, we can reallocate space and rehash.
- Chaining uses an "overflow" list for each element in the hash table.
- When a collision occurs, we can evict the occupant (LCFS), evict the latecomer (FCFS) or evict the element furthest from its home location (Robin Hood hashing).
- When a collision occurs in open addressing, we can
  - probe nearby for a free position (linear probing, quadratic probing);
  - go to a "random" position by using a second-level hash function (double hashing);

Collision resolution policies

- Open addressing uses no extra space - every element is stored in the hash table. If it gets overfull, we can reallocate space and rehash.
- Chaining uses an "overflow" list for each element in the hash table.
- When a collision occurs, we can evict the occupant (LCFS), evict the latecomer (FCFS) or evict the element furthest from its home location (Robin Hood hashing).
- When a collision occurs in open addressing, we can
  - probe nearby for a free position (linear probing, quadratic probing);
  - go to a "random" position by using a second-level hash function (double hashing);
  - try a position given by a second, third, . . . hash function (this plus LCFS gives *cuckoo hashing*).

Collision resolution via chaining

- Elements that hash to the same slot are placed in a list. The slot contains a pointer to the head of this list.

Collision resolution via chaining

- Elements that hash to the same slot are placed in a list. The slot contains a pointer to the head of this list.
- Insertion can then be done in constant time.

Collision resolution via chaining

- Elements that hash to the same slot are placed in a list. The slot contains a pointer to the head of this list.
- Insertion can then be done in constant time.
- Deletion can be done in constant time with a doubly linked list, for example.

Collision resolution via chaining

- Elements that hash to the same slot are placed in a list. The slot contains a pointer to the head of this list.
- Insertion can then be done in constant time.
- Deletion can be done in constant time with a doubly linked list, for example.
- A drawback is the additional space overhead. Also, the distribution of sizes of lists turns out to be very uneven.

Collision resolution via open addressing

- Every key is stored somewhere in the array; no extra space is used.

Collision resolution via open addressing

- Every key is stored somewhere in the array; no extra space is used.
- If a key k hashes to a value $h(k)$ that is already occupied, we probe (look for an empty space).

Collision resolution via open addressing

- Every key is stored somewhere in the array; no extra space is used.
- If a key k hashes to a value $h(k)$ that is already occupied, we probe (look for an empty space).
    - The most common probing method is linear probing, which moves left one index at a time, wrapping around if necessary, until it finds an empty address. This is easy to implement but leads to clustering.

Collision resolution via open addressing

- Every key is stored somewhere in the array; no extra space is used.
- If a key k hashes to a value $h(k)$ that is already occupied, we probe (look for an empty space).
  - The most common probing method is linear probing, which moves left one index at a time, wrapping around if necessary, until it finds an empty address. This is easy to implement but leads to clustering.
  - Another method is double hashing. Move to the left by a fixed step size t, wrapping around if necessary, until we find an empty address. The difference is that t is not fixed in advance, but is given by a second hashing function $p(k)$.

Analysis of hashing

- We count cost by number of key comparisons/probes.

Analysis of hashing

- We count cost by number of key comparisons/probes.
- Insertion has the same cost as an unsuccessful search, provided the table is not full.

Analysis of hashing

- We count cost by number of key comparisons/probes.
- Insertion has the same cost as an unsuccessful search, provided the table is not full.
- The average cost of a successful search is the average of all insertions needed to construct the table from empty.

Analysis of hashing

- We count cost by number of key comparisons/probes.
- Insertion has the same cost as an unsuccessful search, provided the table is not full.
- The average cost of a successful search is the average of all insertions needed to construct the table from empty.
- We often use the simple uniform hashing model. That is, each of the n keys is equally likely to hash into any of the m slots. So we are considering a “balls in bins” model.

Analysis of hashing

- We count cost by number of key comparisons/probes.
- Insertion has the same cost as an unsuccessful search, provided the table is not full.
- The average cost of a successful search is the average of all insertions needed to construct the table from empty.
- We often use the simple uniform hashing model. That is, each of the n keys is equally likely to hash into any of the m slots. So we are considering a “balls in bins” model.
- If n is much smaller than m, collisions will be few and most slots will be empty. If n is much larger than m, collisions will be many and no slots will be empty. The most interesting behaviour is when m and n are of comparable size.

Analysis of hashing

- We count cost by number of key comparisons/probes.
- Insertion has the same cost as an unsuccessful search, provided the table is not full.
- The average cost of a successful search is the average of all insertions needed to construct the table from empty.
- We often use the simple uniform hashing model. That is, each of the n keys is equally likely to hash into any of the m slots. So we are considering a "balls in bins" model.
- If n is much smaller than m, collisions will be few and most slots will be empty. If n is much larger than m, collisions will be many and no slots will be empty. The most interesting behaviour is when m and n are of comparable size.
- Define the load factor to be $\lambda := n/m$.

Analysis of balls in boxes

- Define

$$Q(m,n) = \frac{m!}{(m-n)!m^n} = \frac{m}{m}\frac{m-1}{m}\dots\frac{m-n+1}{m}.$$

Note that $Q(m,0) = 1, Q(m,n) = 0$ unless $0 \leq n \leq m$.

Analysis of balls in boxes

- Define

$$Q(m,n) = \frac{m!}{(m-n)!m^n} = \frac{m}{m}\frac{m-1}{m}\dots\frac{m-n+1}{m}.$$

  Note that $Q(m,0) = 1, Q(m,n) = 0$ unless $0 \leq n \leq m$.
- The probability of no collisions when n balls are thrown into m boxes uniformly at random is $Q(m,n)$. For example, $Q(366,180) \approx 0.4486998183 \times 10^{-23}$, $Q(366,24) \approx 0.4626535709$.

Analysis of balls in boxes

- Define

$$Q(m,n) = \frac{m!}{(m-n)!m^n} = \frac{m}{m}\frac{m-1}{m}\cdots\frac{m-n+1}{m}.$$

  Note that $Q(m,0) = 1, Q(m,n) = 0$ unless $0 \leq n \leq m$.

- The probability of no collisions when n balls are thrown into m boxes uniformly at random is $Q(m,n)$. For example, $Q(366,180) \approx 0.4486998183 \times 10^{-23}$, $Q(366,24) \approx 0.4626535709$.
- The expected number of balls until the first collision is equal to $E(m) := \sum_{n \leq m} Q(m,n)$. Note $E(365) \approx 25$.

Statistics for balls in bins: some facts

- When do we expect the first collision? This is the birthday problem. Answer: $E(m) \approx \sqrt{\pi m/2} + 2/3$. So collisions happen even in fairly sparse tables.

Statistics for balls in bins: some facts

- When do we expect the first collision? This is the birthday problem. Answer: $E(m) \approx \sqrt{\pi m/2} + 2/3$. So collisions happen even in fairly sparse tables.
- When do we expect all boxes to be nonempty? This is the coupon collector problem. Answer: after about $m \log m$ balls. It takes a long time to use all lists when chaining.

Statistics for balls in bins: some facts

- When do we expect the first collision? This is the birthday problem. Answer: $E(m) \approx \sqrt{\pi m/2} + 2/3$. So collisions happen even in fairly sparse tables.
- When do we expect all boxes to be nonempty? This is the coupon collector problem. Answer: after about $m \log m$ balls. It takes a long time to use all lists when chaining.
- What proportion of boxes are expected to be empty when $m \approx n$? Answer: $e^{-\lambda}$. Many of the lists are just wasted space even for pretty full tables.

Statistics for balls in bins: some facts

- When do we expect the first collision? This is the birthday problem. Answer: $E(m) \approx \sqrt{\pi m/2} + 2/3$. So collisions happen even in fairly sparse tables.
- When do we expect all boxes to be nonempty? This is the coupon collector problem. Answer: after about $m \log m$ balls. It takes a long time to use all lists when chaining.
- What proportion of boxes are expected to be empty when $m \approx n$? Answer: $e^{-\lambda}$. Many of the lists are just wasted space even for pretty full tables.
- When $m = n$, what is the expected maximum number of balls in a box? Answer: about $(\log n)/(\log \log n)$. Some of the lists may be fairly long.

Statistics for balls in bins: some facts

- When do we expect the first collision? This is the birthday problem. Answer: $E(m) \approx \sqrt{\pi m/2} + 2/3$. So collisions happen even in fairly sparse tables.
- When do we expect all boxes to be nonempty? This is the coupon collector problem. Answer: after about $m \log m$ balls. It takes a long time to use all lists when chaining.
- What proportion of boxes are expected to be empty when $m \approx n$? Answer: $e^{-\lambda}$. Many of the lists are just wasted space even for pretty full tables.
- When $m = n$, what is the expected maximum number of balls in a box? Answer: about $(\log n)/(\log \log n)$. Some of the lists may be fairly long.
- The analysis that gives these results is beyond this course. See me for references.

Results for chaining

- The worst case for searching is $\Theta(n)$, since there may be only one chain with all the keys.

Results for chaining

- The worst case for searching is $\Theta(n)$, since there may be only one chain with all the keys.
- The average cost for unsuccessful search is the average list length, namely λ.

Results for chaining

- The worst case for searching is $\Theta(n)$, since there may be only one chain with all the keys.
- The average cost for unsuccessful search is the average list length, namely λ.
- The average cost for successful search is then

$$\frac{1}{n}\sum_{k=1}^{n}\frac{k}{m} = \frac{n+1}{2m} \approx \lambda/2.$$

Results for chaining

- The worst case for searching is $\Theta(n)$, since there may be only one chain with all the keys.
- The average cost for unsuccessful search is the average list length, namely λ.
- The average cost for successful search is then

$$\frac{1}{n}\sum_{k=1}^{n}\frac{k}{m} = \frac{n+1}{2m} \approx \lambda/2.$$

- Thus provided the load factor is kept bounded, basic operations run in constant time on average.

Results for open addressing

- We assume uniform hashing: each configuration of n keys in a table of size m is equally likely to occur. In other words, the hash function produces a random permutation of the keys, and the slots are probed in random order for each key.

Results for open addressing

- We assume uniform hashing: each configuration of n keys in a table of size m is equally likely to occur. In other words, the hash function produces a random permutation of the keys, and the slots are probed in random order for each key.
- Uniform hashing can't be implemented practically but seems to be a good model when double hashing is used.

Results for open addressing

- We assume uniform hashing: each configuration of n keys in a table of size m is equally likely to occur. In other words, the hash function produces a random permutation of the keys, and the slots are probed in random order for each key.
- Uniform hashing can't be implemented practically but seems to be a good model when double hashing is used.
- It can be shown that the average cost for insertion under this hypothesis is $\Theta(1/(1-\lambda))$ as $m \to \infty$.

Results for open addressing

- We assume uniform hashing: each configuration of n keys in a table of size m is equally likely to occur. In other words, the hash function produces a random permutation of the keys, and the slots are probed in random order for each key.
- Uniform hashing can't be implemented practically but seems to be a good model when double hashing is used.
- It can be shown that the average cost for insertion under this hypothesis is $\Theta(1/(1-\lambda))$ as $m \to \infty$.
- Linear probing is not well described by the uniform hashing model (because of the clustering). A more detailed analysis can be done. The average insertion cost is $\Theta(1 + 1/(1-\lambda)^2)$.

Results for open addressing

- We assume uniform hashing: each configuration of n keys in a table of size m is equally likely to occur. In other words, the hash function produces a random permutation of the keys, and the slots are probed in random order for each key.
- Uniform hashing can't be implemented practically but seems to be a good model when double hashing is used.
- It can be shown that the average cost for insertion under this hypothesis is $\Theta(1/(1-\lambda))$ as $m \to \infty$.
- Linear probing is not well described by the uniform hashing model (because of the clustering). A more detailed analysis can be done. The average insertion cost is $\Theta(1 + 1/(1-\lambda)^2)$.
- If the load factor is bounded away from 1, basic operations run in constant time; otherwise performance will be very bad.

Results for open addressing

- We assume uniform hashing: each configuration of n keys in a table of size m is equally likely to occur. In other words, the hash function produces a random permutation of the keys, and the slots are probed in random order for each key.
- Uniform hashing can't be implemented practically but seems to be a good model when double hashing is used.
- It can be shown that the average cost for insertion under this hypothesis is $\Theta(1/(1-\lambda))$ as $m \to \infty$.
- Linear probing is not well described by the uniform hashing model (because of the clustering). A more detailed analysis can be done. The average insertion cost is $\Theta(1 + 1/(1-\lambda)^2)$.
- If the load factor is bounded away from 1, basic operations run in constant time; otherwise performance will be very bad.
- See me for details of the proofs.