COMP 3170 - Analysis of Algorithms & Data Structures

Shahin Kamali

Binomial Heaps
CLRS 6.1, 6.2, 6.3
University of Manitoba
Priority queues

- A **priority queue** is an abstract data type formed by a set \( S \) of key-value pairs.

- **Basic operations** include:
  - **insert** \((k)\) inserts a new element with key \( k \) into \( S \).
  - **get-Max** which returns the element of \( S \) with the largest key.
  - **extract-Max** which returns the element of \( S \) with the largest key and delete it from \( S \).

- We are often given the whole data and need to **build** the data structure based on it.
  - Any data structure for a priority queue should be **constructed** efficiently.
What is a good implementation (data structure) for priority queues?

- Binary heaps: get-Max runs in $O(1)$ and extract-Max and insert take $\Theta(\log n)$ for $n$ keys.

- Balanced binary search tree: get-Max runs in $O(1)$ and extract-Max and insert both can run in $\Theta(\log n)$.

The problem with BSTs: it is costly to build them.

How long does it take to form a BST from a given set of items? It takes $\Omega(n \log n)$; otherwise you can sort them in $o(n \log n)$ by building the BST and doing an in-order traverse in $O(n)$.

We know we cannot comparison-sort in $o(n \log n)$ and hence cannot build the tree in such time.
What is a good implementation (data structure) for priority queues?

You have seen **binary heaps** before: get-Max runs in $O(1)$ and extract-Max and insert both take $\Theta(\log n)$ for $n$ keys.
What is a good implementation (data structure) for priority queues?

You have seen binary heaps before: get-Max runs in $O(1)$ and extract-Max and insert both take $\Theta(\log n)$ for $n$ keys.

Is a balanced binary search tree a good implementation of a priority queue?
Priority queue implementation

- What is a good implementation (data structure) for priority queues?
- You have seen **binary heaps** before: get-Max runs in $O(1)$ and extract-Max and insert both take $\Theta(\log n)$ for $n$ keys.
- Is a balanced binary search tree a good implementation of a priority queue?
  - with a little augmentation, get-Max runs in $O(1)$ and extract-Max and insert both can run in $\Theta(\log n)$. 

The problem with BSTs: it is costly to build them.

How long does it take to form a BST from a given set of items?

It takes $\Omega(n \log n)$; otherwise you can sort them in $o(n \log n)$ by building the BST and doing an inoder traverse in $O(n)$.

We know we cannot comparison-sort in $o(n \log n)$ and hence cannot build the tree in such time.
What is a good implementation (data structure) for priority queues?

You have seen **binary heaps** before: get-Max runs in $O(1)$ and extract-Max and insert both take $\Theta(\log n)$ for $n$ keys.

Is a balanced binary search tree a good implementation of a priority queue?

- with a little augmentation, get-Max runs in $O(1)$ and extract-Max and insert both can run in $\Theta(\log n)$.

The problem with BSTs: it is costly to build them

- How long does it take to form a BST from a given set of items?
- It takes $\Omega(n \log n)$; otherwise you can sort them in $o(n \log n)$ by building the BST and doing an inorder traverse in $O(n)$.
- We know we cannot comparison-sort in $o(n \log n)$ and hence cannot build the tree in such time.
Binary heaps

- A **heap** is a **tree** data structure.
- For every node \( i \) other than the root, we have \( key[parent[i]] \geq key[i] \).
- A **binary heap** is a complete binary tree which can be stored using an array.
  - `build-heap` takes \( \Theta(n) \) time
  - `insert`, `extract-Max` take \( \Theta(\log n) \)
  - `get-Max` takes \( O(1) \)
Binary heaps

Suppose multiple priority queues on different servers.

Occasionally a server must be rebooted, requiring two priority queues to be **merged**.

With a typical binary heap, merging requires concatenating arrays and **re-running** build-heap; this takes $\Theta(n)$ :-(

When implementing an abstract data type always consider if you need it to be **mergable** or not.

![Binary heap diagrams](image.png)
Binary heaps

- Suppose multiple priority queues on different servers.
- Occasionally a server must be rebooted, requiring two priority queues to be **merged**.
- With a typical binary heap, merging requires concatenating arrays and **re-running** build-heap; this takes $\Theta(n)$.
- When implementing an abstract data type always consider if you need it to be **mergable** or not.
We would like to build a data structure for priority queues that:

- supports insert, extract-Max, get-Max, and build efficiently (as in binary heaps)
- merging two priority queues takes $o(n)$
Rethinking about Data Structure

- We would like to build a data structure for priority queues that:
  - supports insert, extract-Max, get-Max, and build efficiently (as in binary heaps)
  - merging two priority queues takes $o(n)$

- Solution: binomial heaps which are mergable heaps that efficiently support
  - $\text{insert}(H, x)$
  - $\text{extract-Max}(H)$
  - $\text{get-Max}(H)$
  - $\text{build}(A)$
  - $\text{union}(H_1, H_2)$ (merge)
  - $\text{increase-key}(H, x, k)$
  - $\text{delete}(H, x)$
Bionomial Trees

- A **binomial tree** is an ordered tree defined recursively
  - children of each node have a specific ordering (similar to ‘left’ and ‘right’ child in binary trees).
Bionomial Trees

- A **binomial tree** is an ordered tree defined recursively
  - children of each node have a specific ordering (similar to ‘left’ and ‘right’ child in binary trees).
- The base case for a binomial tree $B_0$ is a single node
- To build $B_k$, we take two copies of $B_{k-1}$ and let the first child of the root of the second copy be the root of the first copy.
A **binomial tree** is an ordered tree defined recursively

- children of each node have a specific ordering (similar to ‘left’ and ‘right’ child in binary trees).

The base case for a binomial tree $B_0$ is a single node

To build $B_k$, we take two copies of $B_{k-1}$ and let the first child of the root of the second copy be the root of the first copy.
Fun with Binomial Trees

Fun 1: The children of the root of the binomial tree $B_k$ are the binomial trees $B_{k-1}, \ldots, B_0$. 
Fun with Binomial Trees

- Fun 1: The children of the root of the binomial tree $B_k$ are the binomial trees $B_{k-1}, \ldots, B_0$.
  - Induction: assume it is true for all binomial trees $B_i$ with $i \leq k - 1$ (base easily holds).
  - The tree $B_k$ has its first child as $B_{k-1}$ (recursive construction).
  - With respect to other children, it is a binomial tree $B_{k-1}$ and hence has children $B_{k-2}, \ldots, B_0$ by induction hypothesis.
Fun with Bionomial Trees

- Fun 2: $B_k$ has $2^k$ nodes:
Fun with Bionomial Trees

Fun 2: $B_k$ has $2^k$ nodes:

- The recursion is $N(B_k) = 2N(B_{k-1})$, $N(B_0) = 1$
Fun with Binomial Trees

- Fun 2: $B_k$ has $2^k$ nodes:
  - The recursion is $N(B_k) = 2N(B_{k-1})$, $N(B_0) = 1$
- $B_k$ has height $k$: 

![Binomial Tree Diagram]
Fun with Binomial Trees

- Fun 2: $B_k$ has $2^k$ nodes:
  - The recursion is $N(B_k) = 2N(B_{k-1})$, $N(B_0) = 1$

- $B_k$ has height $k$:
  - The recursion is $h(B_k) = h(B_{k-1}) + 1$:
Fun with Bionomial Trees

- **Fun 2:** $B_k$ has $2^k$ nodes:
  - The recursion is $N(B_k) = 2N(B_{k-1})$, $N(B_0) = 1$

- $B_k$ has height $k$:
  - The recursion is $h(B_k) = h(B_{k-1}) + 1$:

- Within $B_k$ there are $\binom{k}{i}$ nodes at depth $i$.
  - The recursion is $ch(k, i) = ch(k - 1, i - 1) + ch(k - 1, i)$
  - Solving this recursion gives $ch(k, i) = \binom{k}{i}$. To get an idea of the proof, note that $\binom{k}{i} = \binom{k-1}{i-1} + \binom{k-1}{i}$
A **binomial heap** is a set of binomial trees such that:

- each binomial tree is heap-ordered \((key[parent[i]] \geq key[i])\)
- for each \(k\) there is at most one binomial tree of order \(k\)
Binomial Heaps

Definition

A **binomial heap** is a set of binomial trees such that:

- each binomial tree is heap-ordered \((\text{key}[\text{parent}[i]] \geq \text{key}[i])\)
- for each \( k \) there is at most one binomial tree of order \( k \)
Binomial Heaps

Definition

A **binomial heap** is a set of binomial trees such that:

- each binomial tree is heap-ordered \((key[parent[i]] \geq key[i])\)
- for each \(k\) there is at most one binomial tree of order \(k\)
Binomial Heaps

Definition

A **binomial heap** is a set of binomial trees such that:

- each binomial tree is heap-ordered ($key[parent[i]] \geq key[i]$)
- for each $k$ there is at most one binomial tree of order $k$
Binomial Heaps

Definition

A **binomial heap** is a set of binomial trees such that:

- each binomial tree is heap-ordered \((key[parent[i]] \geq key[i])\)
- for each \(k\) there is at most one binomial tree of order \(k\)
Binomial Heaps

Definition

A **binomial heap** is a set of binomial trees such that:

- each binomial tree is heap-ordered \((key[parent[i]] \geq key[i])\)
- for each \(k\) there is at most one binomial tree of order \(k\)
How many trees are in a binomial heap of $n$ nodes?

Let $x$ be the number of trees. We express the number of nodes as a function of $x$. The number of nodes is minimized when there is one tree of order $i$ for any $i \in [0, x - 1]$ (note that no two trees of same order can exist).

Recall that a binomial tree of order $i$ has $2^i$ nodes. We have

$$n \geq 1 + 2 + \ldots + 2^{x-1} = 2^x - 1,$$

i.e.,

$$x \leq \log(n + 1).$$

A binomial heap storing $n$ keys has at most $\log(n + 1)$ binomial trees.
How many trees are in a binomial heap of $n$ nodes?

- Let $x$ be the number of trees.
- We express the number of nodes as a function of $x$.
- The number of nodes is minimized when there is one tree of order $i$ for any $i \in [0, x - 1]$ (note that no two trees of same order can exist).
  - Recall that a binomial tree of order $i$ has $2^i$ nodes.
  - We have $n \geq 1 + 2 + \ldots + 2^{x-1} = 2^x - 1$, i.e., $x \leq \log(n + 1)$.

A binomial heap storing $n$ keys has at most $\log(n + 1)$ binomial trees.
Finding Max in Binomial Heaps

- For `get-Max()` operation, just follow the links connecting roots of binomial trees
  - The maximum element in all the heap is the max node, hence root, in one of the trees
  - E.g., max in the below heap is \( \text{max}\{11, 99, 40\} = 90 \)
Finding Max in Binomial Heaps

- For get-Max() operation, just follow the links connecting roots of binomial trees
  - The maximum element in all the heap is the max node, hence root, in one of the trees
  - E.g., max in the below heap is max\{11, 99, 40\} = 90
- There are \(\log(n + 1)\) trees and hence the time complexity is \(\Theta(\log n)\).
  - It is a bit worse that \(O(1)\) of get-Max() in binary heaps
Merging of Two Binomial Heaps

- Union operation: we want to merge two heaps of sizes $n_1$ and $n_2$.
  - Similar to merge operation in merge sort, follow the links connecting roots of the heaps, and ‘merge’ them into one list (i.e., one heap).
  - If two trees of same order $i$ are visited, merge them into a binomial tree of order $i + 1$
    - It is possible by the definition of binomial tree.
    - The tree with the smaller key in its root becomes a child of the other tree.
  - Two trees can be merged in $O(1)$.
  - When 3 trees of order $i$, merge the 2 older trees (keep the new one).
Merging of Two Binomial Heaps

- There is an analogy with **binary** addition: add bits and carry
  - Read from the least significant to the most significant bit (right to left)
  - $111 + 011 = 1010$; “1010” means 1 tree of order 3, 0 tree of order 2, 1 tree of order 1, and 0 tree of order 0.
Merge Time Complexity

What is time complexity of merge?

- Each merge operation takes $O(1)$.
- For each tree rank, there will be at most one merge.
- The total time complexity is

$$O(\log(n_1) + \log(n_2)) = O(2 \log(\max\{n_1, n_2\})) = O(\log n)$$

where $n$ is the size after the merge.

It is possible to merge two binomial heaps in $O(\log n)$ where $n$ is the number of keys after the merge.
Merge Time Complexity

What is time complexity of merge?

- Each merge operation takes $O(1)$.
- For each tree rank, there will be at most one merge.
- The total time complexity is
  $$O(\log(n_1) + \log(n_2)) = O(2\log(\max\{n_1, n_2\})) = O(\log n)$$
  where $n$ is the size after the merge.

It is possible to merge two binomial heaps in $O(\log n)$ where $n$ is the number of keys after the merge.
Insert Operation

To insert a new key $x$ to the priority queue:

- Create a new binomial heap of size 1 (order 0) with the new key
- Return the union of the old heap with the new one (e.g., Insert(40))
**Insert Operation**

To insert a new key $x$ to the priority queue:

- Create a new binomial heap of size 1 (order 0) with the new key
- Return the union of the old heap with the new one (e.g., Insert(40))
- The time complexity is similar to merge.

It is possible to insert a new item to a binomial heap in $O(\log n)$, which is as good as binary heaps.
Extract-Max Operation

To extract max, first search and find the maximum.

- Assuming max is in a binomial tree of order $k$, its children are $k$ binomial trees of order $1, 2, \ldots, k-1$
- Delete max and create a new binomial heap formed by these trees.
- Merge the old heap and the new one.
- The time complexity is $O(\log n)$ for finding the max and $O(\log n)$ for merging the two heaps, i.e., $O(\log n)$ in total
Extract-Max Operation

To extract max, first search and find the maximum.

- Assuming max is in a binomial tree of order \( k \), its children are \( k \) binomial trees of order 1, 2, \ldots, \( k - 1 \)
- Delete max and create a new binomial heap formed by these trees.
- Merge the old heap and the new one.
- The time complexity is \( O(\log n) \) for finding the max and \( O(\log n) \) for merging the two heaps, i.e., \( O(\log n) \) in total.

It is possible to extract maximum element in a binomial heap in \( O(\log n) \), which is as good as binary heaps.
Bionomial Heaps Review

- Get-Max can be done in $\Theta(\log n)$ (a bit slower than $\Theta(1)$ of binary heaps).
- Merge can be done in $\Theta(\log n)$ (much better than $\Theta(n)$ of binary heaps).
- Insert and Extract-Max can be done in $\Theta(\log n)$ (similar to binary heaps)
Increase Key

- Increase$(a, x)$: assume you are given a pointer to a key $a$ and want to increase it by $x$. 

Time is proportional to the height of a binomial tree, i.e., the order of the tree. Recall that a binomial tree of order $k$ has $2^k$ nodes, so, the order and hence the height of any tree in the heap is $O(\log n)$. Increase the key of a given node can be done in time $\Theta(\log n)$. 

- Note that if the pointer is not given, you need to search for the key, which takes $\Theta(n)$ in any heap (heaps are NOT good for searching). Increase the key and 'float' it upward until key[$parent[i]$] ≥ key[i] (e.g., increase '8' to '68').
Increase Key

- Increase(a, x): assume you are given a pointer to a key a and want to increase it by x.
  - Note that if the pointer is not given, you need to search for the key, which takes $\Theta(n)$ in any heap (heaps are NOT good for searching).
**Increase Key**

- **Increase**($a,x$): assume you are given a pointer to a key $a$ and want to increase it by $x$.
  - Note that if the pointer is not given, you need to search for the key, which takes $\Theta(n)$ in any heap (heaps are NOT good for searching)
- Increase the key and ‘float’ it upward until $\text{key}[\text{parent}[i]] \geq \text{key}[i]$ (e.g., increase ’8’ to ’68’).
Increase Key

- Increase($a,x$): assume you are given a pointer to a key $a$ and want to increase it by $x$.
  - Note that if the pointer is not given, you need to search for the key, which takes $\Theta(n)$ in any heap (heaps are NOT good for searching).
- Increase the key and ‘float’ it upward until $key[parent[i]] \geq key[i]$ (e.g., increase '8' to '68').
Increase Key

- Increase($a, x$): assume you are given a pointer to a key $a$ and want to increase it by $x$.
  - Note that if the pointer is not given, you need to search for the key, which takes $\Theta(n)$ in any heap (heaps are NOT good for searching).
- Increase the key and ‘float’ it upward until $key[parent[i]] \geq key[i]$ (e.g., increase ‘8’ to ‘68’).
**Increase Key**

- Increase\((a, x)\): assume you are given a pointer to a key \(a\) and want to increase it by \(x\).
  - Note that if the pointer is not given, you need to search for the key, which takes \(\Theta(n)\) in any heap (heaps are NOT good for searching).
- Increase the key and ‘float’ it upward until \(key[parent[i]] \geq key[i]\) (e.g., increase '8' to '68').
Increase Key

- Increase($a, x$): assume you are given a pointer to a key $a$ and want to increase it by $x$.
  - Note that if the pointer is not given, you need to search for the key, which takes $\Theta(n)$ in any heap (heaps are NOT good for searching)
- Increase the key and ‘float’ it upward until $key[parent[i]] \geq key[i]$ (e.g., increase ’8’ to ’68’).
- Time is proportional to the height of a binomial tree, i.e., the order of the tree
  - Recall that a binomial tree of order $k$ has $2^k$ nodes, so, the order and hence the height of any tree in the heap is $O(\log n)$.
- **Increase the key of a given node can be done in time $\Theta(\log n)$**.
Delete

- \textbf{Delete}(a): assume you are given a pointer to a key \textit{a} and want to delete it
Delete

Delete\((a)\): assume you are given a pointer to a key \(a\) and want to delete it

- Call Increase-key to set the key to \(\infty\).
- Call Extract-Max to remove the largest item; this would remove our node from the heap

Time is \(O(\log n)\) for Increase-key and \(O(\log n)\) for Extract-Max.
Delete

Delete(a): assume you are given a pointer to a key a and want to delete it

- Call Increase-key to set the key to $\infty$.
- Call Extract-Max to remove the largest item; this would remove our node from the heap

Time is $O(\log n)$ for Increase-key and $O(\log n)$ for Extract-Max.
Delete

Delete(a): assume you are given a pointer to a key a and want to delete it

- Call Increase-key to set the key to $\infty$.
- Call Extract-Max to remove the largest item; this would remove our node from the heap

- Time is $O(\log n)$ for Increase-key and $O(\log n)$ for Extract-Max.
Delete

- **Delete***(a)**: assume you are given a pointer to a key *a* and want to delete it
  - Call **Increase-key** to set the key to $\infty$.
  - Call **Extract-Max** to remove the largest item; this would remove our node from the heap

- Time is $O(\log n)$ for **Increase-key** and $O(\log n)$ for **Extract-Max**.
Delete

Delete(a): assume you are given a pointer to a key a and want to delete it

- Call Increase-key to set the key to $\infty$.
- Call Extract-Max to remove the largest item; this would remove our node from the heap

Time is $O(\log n)$ for Increase-key and $O(\log n)$ for Extract-Max.

Deleting a given node can be done in time $O(\log n)$. 
Binomial Heaps Summary

Given a key (a pointer to its node), we can increase or delete that node in $O(\log n)$.

**Theorem**

Priority queries can be implemented with binomial tree so that Get-Max, Merge, Extract-Max, Increase (with given pointer) and delete (with given pointer) can all be performed in $O(\log n)$. 

COMP 3170 - Analysis of Algorithms & Data Structures