Design and Analysis of Algorithms II
Lecture 2 Heaps and Dynamic Arrays

Gordon Lu (gl1589@nyu.edu)

September 14, 2023

1 Dynamic Arrays

1.1 Dynamic Arrays: Last Time

Recall our setup in lecture 1. We have an array with the following parameters:

S	Array Size (capacity)
n	# of Stored Elements

We assumed that we could insert a **theoretically** endless amount of items into the array without repercussions. But what if we inserted 1,000,000 items, and then deleted 999,999 of those items?

- We would have 1 element, but 999,999 vacant slots!

1.2 Dynamic Arrays: Setup

In order to limit the amount of wasted memory, we introduce the idea of a **load factor**. We define it as $\alpha = n/S$, this is simply how full the array is with respect to its capacity.

Our idea with using the load factor is to keep α within some constant limits, otherwise we will either have to shrink or double. Specifically, for some constants c_1, c_2, we want $c_1 \leq \alpha \leq c_2$.

We know that if we only had insertions, the amortized run time would be $O(1)$ where $\alpha \leq \frac{1}{2}$.

- If α falls below some threshold, we shrink.

1.3 Dynamic Array Algorithm

Algorithm 1 Dynamic Arrays

if $\alpha = \frac{1}{2}(full)$ **then** Double the array
else if $\alpha < \frac{1}{4}$ **then** Shrink the array

1.4 Dynamic Array: Amortized Analysis

Similar to our analysis with a dynamic array only supporting inserts, we must now provide an amortized analysis for a dynamic array supporting deletes and inserts.

We will use the potential method to analyze the cost of a sequence of n insert and delete operations. The potential function we will use is:

$$\Phi = \left\{ \begin{array}{ll} 2n - S, & \text{if } \alpha \geq \frac{1}{2} \\ \frac{S}{2} - n, & \text{if } \alpha < \frac{1}{2} \end{array} \right\}$$

Similar to how insert was analyzed in two cases, the potential function has four cases:

- Insertion
- Insertion that triggers doubling
- Deletion
- Deletion that triggers shrinking

The textbook has an in-depth computation of the amortized cost in each case (section 17.4.2), therefore it will be omitted from the notes.

Observe that the potential of an empty table is 0, and the potential is never negative. Thus, the total amortized cost of a sequence of operations with respect to Φ provides an upper bound on the actual cost of the sequence.

2 Priority Queues (PQ)

In order to prepare for the new data structure that will discussed in the next section, we will refresh our knowledge with PQs. For simplicity, we will just discuss min-PQs here.

2.1 Definition

A **priority queue** is a data structure for maintaining a set S of nodes, each with an associated value called a *key*.

2.2 Min-PQ Operations

Note that PQs are not optimized for searching.
Thus, we will need an indirect pointer to where the item is, as we cannot search quickly for nodes with a given key.
A **min-PQ** supports the following operations:

$INSERT(S, x)$	inserts the node x into S.
$EXTRACT\text{-}MIN(S)$	removes & returns the node with the smallest key.
$FIND\text{-}MIN(S)$	identifies the node with the smallest key.
$DECREASE\text{-}KEY(S, x, newKey)$	decreases the value of node x's key to the new value, $newKey$.
$DELETE(S, x)$	deletes node x from S

For a refresher on how these operations work, Section 6.5 in the textbook provides a section on **max-PQs**.

2.3 Heap Examples & Drawback

Priority Queues are internally represented using a heap, usually a binary heap, which has $O(\log n)$ for all operations except $FIND\text{-}MIN$, where n is the current number of elements.
Common applications of Priority Queues include:

- Prim's MST Algorithm
- Dijkstra's Shortest Path Algorithm

Binary heaps tend to have good runtime with most operations, however for the **UNION** operation that merges two heaps, it has runtime $\theta(n)$.

The most natural question would be... **Can we do better?**

- This leads us to our discussion of **Binomial Heaps!**

3 Binomial Trees

Similar to with binary heaps, binomial heaps are inefficient with $SEARCH$, thus we pass a node as part of the input to some functions.
Binomial heaps are a **mergeable heap**, which support the following operations:

$MAKE\text{-}HEAP()$	creates and returns a new heap containing no elements.
$INSERT(H, x)$	inserts node x, whose key has already been filled in, into H.
$FIND\text{-}MIN(H)$	returns a pointer to the node in H whose key is minimum.
$EXTRACT\text{-}MIN(H)$	deletes the node whose key is min, returning pointer to the node.
$UNION(H_1, H_2)$	create & return a heap that contains all nodes of H_1 & H_2.
$DECREASE\text{-}KEY(H, x, newKey)$	assigns node x within heap H the new key, $newKey$.
$DELETE(H, x)$	deletes node x from heap H.

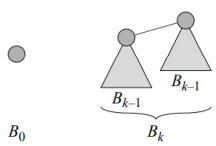
Note that in $UNION$, H_1 and H_2 are destroyed in the process. For a binomial heap, all operations will run in $O(\log n)$, which we will show soon.

3.1 Definition

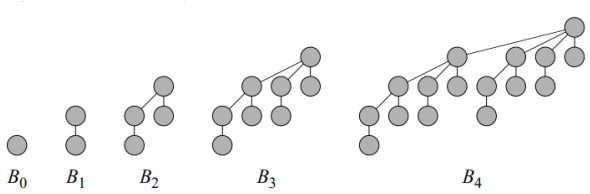
A **binomial heap** is a collection of binomial trees. Therefore, we will define binomial trees and prove key properties.

3.2 Binomial Trees: Definition

A **binomial tree** B_k is an ordered tree defined recursively.
The binomial tree B_k consists of two binomial trees B_{k-1} that are **linked together**: the root of one is the **leftmost child** of the root of the other.



If we consider the number of nodes at each level top-down, we will get the **binomial coefficients**, hence why these are called **binomial trees**. In fact, the terms $\binom{k}{i}$ from **Property 3** are binomial coefficients.



3.3 Binomial Trees: Properties

For the binomial tree B_k,

1. There are 2^k nodes
2. The height of the tree is k
3. There are exactly $\binom{k}{i}$ nodes at depth i for $i = 0, 1, \ldots, k$
4. The root degree is k (degree is the number of children), which is the highest degree node in the tree.
    - The degree of children are $(k-1), (k-2), \ldots$ in order

Proof: We will use induction to prove each property. For each property, the basis is B_0. Verifying that each property holds for B_0 is trivial. For the inductive step, assume that the properties hold for B_{k-1}.

1. Binomial tree B_k consists of two copies of B_{k-1}, so B_k has $2 \times 2^{k-1} = 2^k$ nodes.
2. From the way in which the two copies of B_{k-1} are linked to form B_k, the maximum depth of a node in B_k is one greater than the maximum depth in B_{k-1}. By the inductive hypothesis, this maximum depth is $(k-1)+1 = k$.
3. Let $D(k, i)$ be the number of nodes at depth i of binomial tree B_k. Since B_k is composed of two copies of B_{k-1} linked together, a node at depth i in B_{k-1} appears in B_k once at depth i and once at depth $(i+1)$. In other words, the number of nodes at depth i in B_k is the number of numbes at depth i in $B_{k-1}+$ the number of nodes at depth $(i-1)$ in B_{k-1}. Therefore,
   $D(k, i) = D(k-1, i) + D(k-1, i-1)$
   $= \binom{k-1}{i} + \binom{k-1}{i-1}$
   $= \binom{k}{i}$
4. The only node with greater degree in B_k than in B_{k-1} is the root, which has one more child than in B_{k-1}. Since the root of B_{k-1} has degree $(k-1)$, the root of B_k has degree k. By the inductive hypothesis, the children of the root of B_{k-1} are roots of B_{k-2}, $B_{k-3}, \ldots B_0$. When B_{k-1} is linked to B_{k-1}, the children of the resulting roots are roots of B_{k-1}, $B_{k-2}, \ldots B_0$ which will have degree $(k-1), (k-2), \ldots$

4 Binomial Heaps

4.1 Binomial Heap Properties

A **binomial heap** H is a set of binomial trees that satisfies the following properties:

1. Each binomial tree in H obeys the **min-heap property**: $key(node) \geq key(parent)$. Each such tree is **min-heap-ordered**.
2. No two trees have the same degree
3. The roots of the binomial trees within a binomial heap are organized in a linked list, called the **root list**.
    - The degrees of the roots strictly increase as we traverse the list. (The heap is ordered by degree)

The root of a min-heap-ordered tree contains the smallest key in the tree, therefore the **minimum key** must reside in the root list.

The binary representation of the total number of nodes in the binomial heap will tell us which binomial trees are present. For example, consider a binomial heap H with 13 nodes. The binary representation of 13 is $\langle 1101 \rangle$, and H consists of min-heap-ordered binomial trees B_3, B_2, B_0, having 8, 4 and 1 nodes for a total of 13 nodes.

From this, we can conclude binomial heap H contains at most $\lfloor \log n \rfloor + 1$ binomial trees, as $\lfloor \log n \rfloor + 1$ is the number of bits required to represent n, which is also the number of binomial trees that H will contain.

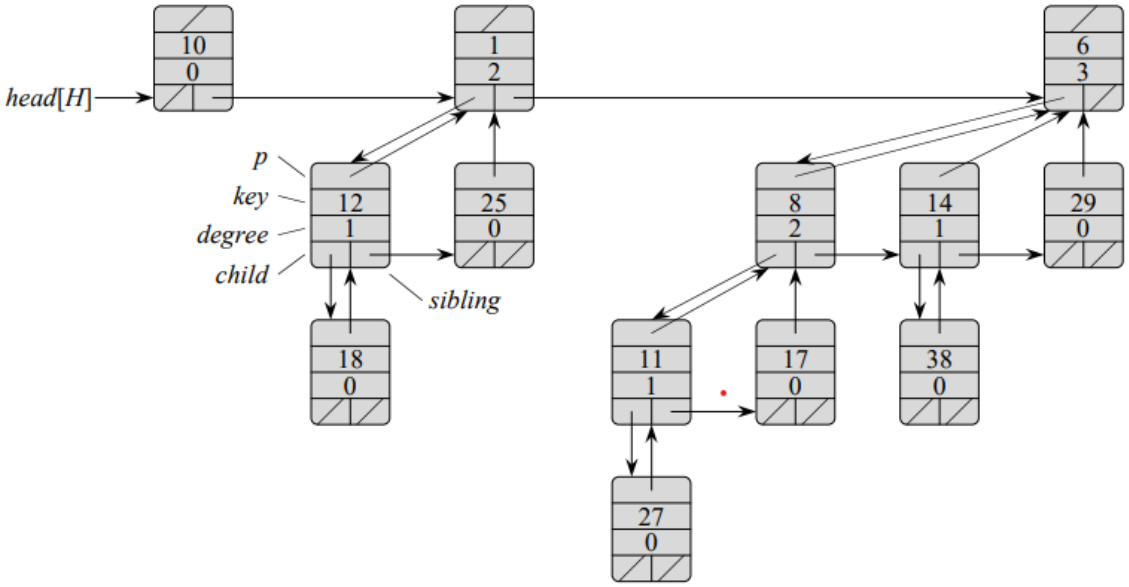
4.2 Representing Binomial Heaps

Each node in a binomial tree within a binomial heap is stored in a **left-child, right-sibling** representation.
Each node will contain the following:

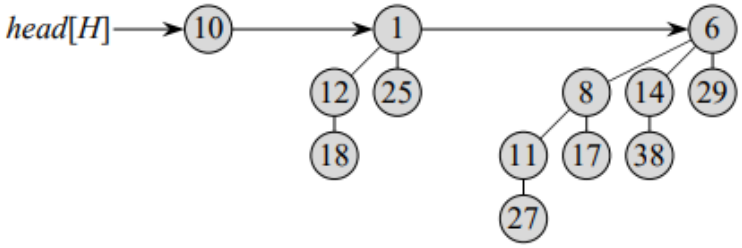
key	key containing data
p[x]	pointer to x's parent
child[x]	pointer to x's leftmost child
sibling[x]	pointer to x's immediate right sibling
degree[x]	number of children of x

1. If node x has no children, then $child[x]$ =NIL.
2. If x is the rightmost child of its parent, then $sibling[x]$ =NIL.
3. If x is a root, then $sibling[x]$ points to the next root in the root list.

Below is a diagram of a detailed representation of a binomial heap H:



Here is a more simplified representation of H:



Observe that for a given binomial heap H, it is accessed by the field $head[H]$, which is simply a pointer to the first root in the root list of H.

- If H has no elements, $head[H]$ =NIL

5 Operations on Binomial Heaps

5.1 Creating a new binomial heap

To make an empty binomial heap, **MAKE-BINOMIAL-HEAP** allocates and returns an object H, where $head[H] = \text{NIL}$. The running time is $\Theta(1)$.

5.2 Finding the minimum key

BINOMIAL-HEAP-MINIMUM returns a pointer to the node with the minimum key in an n-node binomial heap H. We simply check all roots (as the binomial heap is min-heap-ordered so the minimum key must reside in a root node), which will be at most $\lfloor \log n \rfloor + 1$. Therefore, the running time **BINOMIAL-HEAP-MINIMUM** is $O(\log n)$.

5.3 Uniting two binomial heaps

BINOMIAL-HEAP-UNION(H_1, H_2) joins two binomial heaps by repeatedly linking binomial trees with roots that have the same degree. It does so by making use of the following subroutines:

- **BINOMIAL-LINK**(x, y) links two binomial trees by simply making node y the parent of node x and adjusting the pointers to comply with the left-child, right-sibling representation. This has running time $O(1)$, as we're just adjusting pointers.
- **BINOMIAL-HEAP-MERGE**(H_1, H_2) merges the root lists of H_1 and H_2 into a single linked list H that is sorted by degree into monotonically increasing order. As each binomial heap will have at most $\lfloor \log n \rfloor + 1$ roots, the running time will be $O(\log n)$.

BINOMIAL-HEAP-UNION(H_1, H_2) will first merge the root lists through **BINOMIAL-HEAP-MERGE**(H_1, H_2), then link roots of equal degree using **BINOMIAL-LINK**(x, y) until at most one root remains of each degree. The specific cases that arise are demonstrated in the textbook in diagrams 19.6(a)-19.6(d).

5.4 Inserting a node

BINOMIAL-HEAP-INSERT(H, x) inserts node x into H by making a one-node binomial heap H' in $O(1)$ using **MAKE-BINOMIAL-HEAP** and uniting it with the n-node binomial heap H in $O(\log n)$ time using **BINOMIAL-HEAP-UNION**(H, H').

5.5 Extracting the node with the minimum key

BINOMIAL-HEAP-EXTRACT-MIN(H) extracts the node with the minimum key from H and returns a pointer to the extracted node by:

1. Finding the root x with the minimum key in the root list of H, and removing x from the root list.
2. Let H' be a new binomial heap made using **MAKE-BINOMIAL-HEAP**

3. Reverse the order of the linked list of x's children, setting the parent pointer of each child to NIL, and $head[H']$ to point to the head of the resulting list.
    - We reverse the list as the binomial tree is ordered from largest degree to smallest.
4. We then merge the two heaps using **BINOMIAL-HEAP-UNION**(H, H').

The running time of **BINOMIAL-HEAP-EXTRACT-MIN**(H) is $O(\log n)$ due to **BINOMIAL-HEAP-UNION**(H, H').

5.6 Decreasing a key

BINOMIAL-HEAP-DECREASE-KEY$(H, x, newKey)$ decreases the key of a node x in H to a new value $newKey$. It operates in a similar way a binary heap's $DECREASE\text{-}KEY$, we bubble up the key in the heap until $key[y] \geq key[z]$ for node y and parent z.

The running time of **BINOMIAL-HEAP-DECREASE-KEY**$(H, x, newKey)$ is $O(\log n)$.

5.7 Deleting a key

BINOMIAL-HEAP-DELETE(H, x) deletes node x from H by:

1. Making x have the unique minimum key using **BINOMIAL-HEAP-DECREASE-KEY**$(H, x, -\infty)$
2. Deleting x from H by using **BINOMIAL-HEAP-EXTRACT-MIN**(H).

The running time of **BINOMIAL-HEAP-DELETE**(H, x) is $O(\log n)$.

6 Fibonacci Heaps

A **Fibonacci heap** is similar to a binomial heap as it not only supports the same operations as a binomial heap, but it is also a collection of trees.

However, unlike a binomial heap, Fibonacci heaps were designed with amortized analysis in mind. Therefore, its behavior is amortized. Fibonacci heaps also have a much more relaxed structure than binomial heaps, which we will soon see.

The cost of all operations is $O(1)$ amortized with the exception of **EXTRACT-MIN** and **DELETE**.

6.1 Structure of Fibonacci Heaps

A **Fibonacci heap** is a collection of min-heap-ordered trees. However, the trees in a Fibonacci heap are not constrained to be binomial trees.

Unlike trees within binomial heaps, which are ordered, trees within Fibonacci heaps are rooted **but unordered**. The order of the degrees of the roots are also arbitrary.

Each node x contains:

1. A pointer $p[x]$ to its parent
2. A pointer $child[x]$ to any one of its children
3. The number of children in the child list $degree[x]$ of x
4. Whether x has lost a child since the last time x was made the child of another node, given by $mark[x]$.
    - Newly created nodes are unmarked, and x becomes unmarked whenever it is made the child of another node. This only comes into significance with **DECREASE-KEY**.

The children of x are linked together in a circular, doubly linked list, called the **child list** of x. Each child y in a child list has pointers **left[y]** and **right[y]** that point to y's left and right siblings.

- If y is an only child, $left[y] = right[y] = y$.

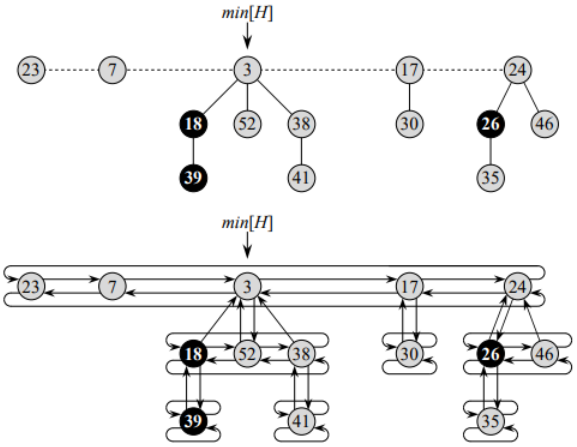
There are two advantages of using circular, doubly linked lists:

1. We can remove a node in $O(1)$
2. Given two such lists, we can concatenate them into one circular, doubly linked list in $O(1)$.

A given Fibonacci heap H is accessed by a pointer $min[H]$ to the root of a tree containing a minimum key. This node is called the **minimum node**.

The roots of all the trees in a Fibonacci heap are linked together using their $left$ and $right$ pointers into a circular, doubly linked list called the **root list**. We also keep track of $n[H]$, the number of nodes currently in H.

Below is a diagram of a Fibonacci Heap.



6.2 Potential Function

We will need to analyze the perform the performance of Fibonacci heaps. As Fibonacci have amortized behavior, we will use the potential method.
For a given Fibonacci heap H, we define the potential to be:

$\Phi(H) = t(H) + 2m(H)$

where $t(H)$ is the number of trees in the root list of H and $m(H)$ is the number of marked nodes in H.
The amortized analyses that will be performed assume that $D(n)$, the maximum degree of any node in an n-node Fibonacci heap has a known upper bound.

- When only **mergeable** heap operations are supported, $D(n) \leq \lfloor \log n \rfloor$
- When DECREASE-KEY and DELETE are also supported, $D(n) = O(\log n)$

6.3 Mergeable-heap operations

If only MAKE-HEAP, INSERT, FIND-MIN, EXTRACT-MIN, and UNION are supported, then each Fibonacci heap is simply a collection of unordered binomial trees.
An **unordered binomial tree** is like a binomial tree, and is also defined recursively. The same properties that hold for binomial trees also hold for unordered binomial trees, with the following variation to **Property 4**:

4'. For the unordered binary tree U_k, the root has degree k, which is greater than that of any other node. The children of the root are roots of subtrees $U_0, U_1, \ldots, U_{k-1}$ in some order.

Thus, if an n-node Fibonacci heap is a collection of unordered binomial trees then $D(n) = \lfloor \log n \rfloor$.
Key idea: The key idea in the mergeable heap operations on Fibonacci heaps is to delay work as long as possible. We rely on **EXTRACT-MIN** to do a lot of the heavy lifting.

6.4 Creating a new Fibonacci heap

MAKE-FIB-HEAP allocates and returns a Fibonacci heap object, and is $O(1)$. Thus, the amortized cost is equal to its actual cost.

6.5 Inserting a node

FIB-HEAP-INSERT(H, x) inserts node x into H by:

1. Making x its own circular, doubly linked list
2. Adds it to the root list of H.
3. Update $min[H]$ if necessary

The change in potential is:
$\Delta\Phi = (t(H') + 2m(H')) - (t(H) + 2m(H))$
$= ((t(H) + 1) + 2m(H)) - (t(H) + 2m(H)) = 1$
Since the actual cost is $O(1)$, the amortized cost is $O(1) + 1 = O(1)$.

6.6 Finding the minimum node

The minimum node of H is given by $min[H]$, so we can find the minimum in $O(1)$. The potential of H does not change, thus the amortized cost is the same as the actual cost, $O(1)$.

6.7 Uniting two Fibonacci heaps

FIB-HEAP-UNION(H_1, H_2) unites H_1 and H_2 by simply concatenating the root lists of H_1 and H_2 and then determining the new minimum node.
The change in potential is:
$\Delta\Phi = \Phi(H) - (\Phi(H_1) + \Phi(H_2)$
$= (t(H) + 2m(H)) - ((t(H_1) + 2m(H_1)) + (t(H_2) + 2m(H_2))) = 0$
since $t(H) = t(H_1) + t(H_2)$ and $m(H) = m(H_1) + m(H_2)$. The amortized cost is therefore the same as the actual cost, $O(1)$.

6.8 Extracting the minimum node

We first introduce the idea of **CONSOLIDATING**. This simply means modifying the root list until every root in the root list has **distinct degrees**.

Recall that in binomial heaps, consolidating is done with the **BINOMIAL-HEAP-UNION** operation, which is done with almost every operation. However with Fibonacci heaps, we delay the work to be done with **FIB-HEAP-EXTRACT-MIN**.
FIB-HEAP-EXTRACT-MIN will:

1. Make a root out of each of the minimum node's children and glue it to the root list (while removing the minimum node.
2. Then consolidate the root list by linking roots of equal degree until at most one root remains of each degree.

More details will be discussed next lecture.

7 Additional Resources

Although I heavily relied on my notes for these scribe notes, I also used the textbook[1] heavily to add more details to what was said in class. Most proofs are taken from the book, as well as some additional information about how certain operations work

References

[1] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.