Part II. ADT, Priority Queues, and Heaps

CS240E: Data Structures and Data Management (Enriched)

2019 Winter, David Duan

Contents

1 Abstract Data Types

1.1 ADT

- **Abstract Data Type**: A description of *information* and a collection of *operations* on that information.
- The information is accessed *only* through the operations.
- We can have various *realizations* of an ADT, which specify:
    - How the information is stored -- *data structure*
    - How the operations are performed -- *algorithms*

1.2 Stack ADT

- **Stack**: An ADT consisting of a collection of items with operations:
    - `push`: inserting an item
    - `pop`: removing the *most* recently inserted item
    - `size`, `isEmpty`, `top`.
- Items are removed in *Last-In First-Out* order.
- Applications: procedure calls, web browser back button.
- Realizations: *arrays* or *linked lists*.

1.3 Queue ADT

- **Queue**: An ADT consisting of a collection of itesm with operations:
    - `enqueue`: inserting an item
    - `dequeue`: removing the *least* recently inserted item
    - `size`, `isEmpty`, `front`.
- Items are removed in *First-In First-Out* order.
- Items enter the queue at the *rear* and are removed from the *front*.
- Applications: CPU scheduling, disk scheduling
- Realizations: *(circular) arrays, linked lists*.

2 Priority Queue

- **PQ**: An ADT consisting of a collection of items (each having a *priority* or *key*) with operations
    - `insert`: inserting an item tagged with a priority
    - `deleteMax` (or `extractMax`, `getMax`): removing the item of *highest priority*
- The above definition is for a *maximum-oriented* priority queue. A *minimum-oriented* priority queue is defined in the natural way, by replacing the operation *deleteMax* by *deleteMin*.
- Applications: to-do list, simulation systems, sorting.

2.1 Using a Priority Queue to Sort

```
PQ-Sort(A[0..n-1])
1.  initialize PQ to an empty priority queue
2.  for k <- 0 to n-1 do
3.    PQ.insert(A[k])
4.  for k <- n-1 down to 0 do
5.    A[k] <- PQ.deleteMax()
```

- Main idea: Insert all the elements to be sorted into a priority queue, and sequentially remove them; they will come out in sorted order.
- Runtime depends on how we implement the priority queue.
- Sometimes written as $O(n + n \cdot \text{insert} + n \cdot \text{deleteMax})$

2.2 Realizations of Priority Queues

2.2.1 Unsorted Arrays / Unsorted Linked Lists

- We assume *dynamic arrays* (amortized over all insertions this takes $O(1)$ extra time).
- $O(1)$ `insert` and $O(n)$ `deleteMax`.
- This realization used for sorting yields *selection sort*.

2.2.2 Sorted Arrays / Sorted Linked Lists

- $O(n)$ `insert` and $O(1)$ `deleteMax`.
- This realization used for sorting yields *insertion sort*.

3 Heaps

A **heap** is a *tree*-based data structure satisfying two properties:

1. The tree is **complete**: all the levels of heap are completely filled, except (possibly) for the last level; the filled items in the level are *left-justified*.
2. The tree satisfies the **heap property**: for any node i, the key of its parent is larger than or equal to the key of i (in a max-heap).

Remarks

1. In a heap, the highest (or lowest) priority element is always stored at the root.
2. A heap is not a softed structure; it can be regarded as being partially sorted.
3. A binary heap with n nodes has height $\Theta(\log n)$.

3.1 Representation

We usually do not store heaps as binary trees but *structured arrays*.

- The *root* is stored at $A[0]$.
- The *children* of $A[i]$ is stored at $A[2i+1]$ and $A[2i+2]$.
- The *parent* of $A[i], i \neq 0$ is $A[\lfloor \frac{i-1}{2} \rfloor]$.
- The *last* node is $A[n-1]$.

We should hide implementation details using helper functions such as `root(), parent(i), last(n), hasLeftChild(i)`, etc.

3.2 Operations on Heaps

3.2.1 Insertion

Place the new key at the first free leaf, then perform a `fix-up`:

```
fix-up(A, k)
k: an index corresponding to a node of the heap

1.  while parent(k) exists and A[parent(k)] < A[k] do
2.    swap A[k] and A[parent(k)]
3.    k <- parent(k)
```

The new item bubbles up until it reaches its correct place in the heap. Insertion has runtime $O(h) = O(\log n)$.

3.2.2 Delete Max

We replace root, which is the max item of the heap, by the last leaf and remove last leaf, then perform a `fix-down`:

```
fix-down(A, n, k)
A: an array that stores a heap of size n
k: an index corresponding to a node of the heap

1.  while k is not a leaf do
2.    // Find the child with the larger key
3.    j <- left child of k
4.    if (j is not last(n) and A[j+1] > A[j])
5.      j <- j + 1
6.    if A[k] >= A[j] break
7.    swap A[j] and A[k]
8.    k <- j
```

The new root bubbles down until it reaches its correct place in the heap. DeleteMax has runtime $O(h) = O(\log n)$.

3.2.3 Priority Queue Realization Using Heaps

We can store items in a PQ in array A and keep track of its *size*:

```
insert(x)

1.  increase size
2.  l <- last(size)
3.  A[l] <- x
4.  fix-up(A, l)
```

```
def last(n):
    """Return the index of last item given a n-item heap."""
    return n - 1

def insert(x):
    """Insert item x to the heap."""
    size += 1          # Increase size
    idx = last(size)     # Get index of first free leaf, which is n-1
    A[idx] = x           # Insert x to this position
    fix_up(A, idx)       # Perform fix_up to get it to the correct position.
```

```
deleteMax()

1.  l <- last(size)
2.  swap A[root()] and A[l]
3.  decrease size
4.  fix-down(A, size, root())
5.  return (A[l])
```

```
def root():
    """Return the index of the root of a heap."""
    return 0

def deleteMax():
    """Pop max of a heap"""
    idx = last(size)          # Get position of the last element
    A[root()], A[idx] = A[idx], A[root()]   # Swap last leaf with root
    size -= 1                 # Update size
    fix_down(A, size, root()) # Perform fix_down to get new root to correct position
    return (A[l])             # Return max
```

3.3 Sorting using Heaps

3.3.1 HeapSort

Recall that any PQ can be used to *sort* in time $O(n + n \cdot \text{insert} + n \cdot \text{deleteMax})$. Using the binary-heaps implementation of PQs, we obtain:

```
PQ-SortWithHeaps(A)

1.  initialize H to an empty heap
2.  for k <- 0 to n-1 do
3.    H.insert(A[k])
4.  for k <- n-1 down to 0 do
5.    A[k] <- H.deleteMax()
```

Since both operations run in $O(\log n)$ for heaps, PQ-Sort using heaps takes $O(n \log n)$ time.

We can improve this with two simple tricks:

1. Heaps can be built faster if we know all input in advance.
2. We can use the same array for input and heap, so $O(1)$ additional space. This is called `Heapsort`.

3.3.2 Building Heaps by Bubble-up

Problem: Given n times in $A[0, \ldots, n-1]$, build a heap containing all of them.

Solution 1: Start with an empty heap and inserting item one at a time. This corresponds to doing n fix-ups, so worst-case $\Theta(n \log n)$.

3.3.3 Building Heaps by Bubble-down

Problem: Given n times in $A[0, \ldots, n-1]$, build a heap containing all of them.

Solution 2: Using `fix-downs` instead:

```
heapify(A)
A: an array

1.  n <- A.size()
2.  for i <- parent(last(n)) downto 0 do
3.    fix-down(A, n, i)
```

A careful analysis yields a worst-case complexity of $\Theta(n)$, which means a heap can be built in linear time.

3.3.4 HeapSort Implementation

Idea: PQ-Sort with heaps

Improvement: use same input array A for storing heap.

```
HeapSort(A, n)

1.  //heapify
2.  n <- A.size()
3.  for i <- parent(last(n)) downto 0 do
4.    fix-down(A, n, i)
5.  // repeatedly find maximum
6.  while n > 1
7.    // do deleteMax
8.    swap items at A[root()] and A[last(n)]
9.    decrease n
10.   fix-down(A, n, root())
```

The `for`-loop takes $\Theta(n)$ time and the while-loop takes $O(n \log n)$ time.

3.4 Other Heap Operations

3.4.1 GetMax

- Output the maximum element of the heap, but don't remove it.
- $O(1)$ time.

3.4.2 ChangePriority

```
changePrority(item, newKey)
- Change the priority if <item> to <newKey>. O(log n) time.
- Helper: fix-up(item): if the item's priority is greater than its parent, switch. In
the worst case we need to do log(n) comparisons, so runtime is O(log n).
- Helper: fix-down(item): if the item's priority is less than its children, switch.
For the same reason, runtime is O(log n).

1. key(item) <- newKey    // Set item's key to newKey
2. fix-up(item)           // Bubble up/down the item until it's at the
3. fix-down(item)         // ... correct place
```

- Change the key/priority of an element.
- $O(\log n)$ time.

3.4.3 Delete

```
delete(item)
- Delete the item from an addressable max-heap. O(log n) time.
- As a remark, `infinity` here represents an arbitrarily large value.
- Helper: changePriority - See above, O(log n)
- Helper: deleteMax - See above, O(log 1)

1. changePriority(item, infinity)  // Set item's key to a large number
2. deleteMax()                     // Since changePriority makes sure that the
                                   // ... item is at the correct place, which,
                                   // ... given its new priority is infinity,
                                   // ... it must be the root of the heap.
                                   // ... Thus, deleteMax() removes it.
```

- Now assume that the item knows where it is (aka a *handle*). More on Addressable Heap.
- $O(\log n)$ time.
- *Remark.* Without handle, a missing key woudl lead to worst-case $O(n)$ time.

4 Heap Merge/Join

Problem `join(H1, H2)`: create a heap that stores items of both H_1 and H_2.

- The task is easy for arrays/lists: $O(n)$. (Khan Academy: Linear Time Merging).
- It is also easy for binary heaps: $O(k \log n)$.
    - Suppose we have H_1 and H_2 with $n = |H_1|$ and $k = |H_2|$. WLOG, suppose $k \leq n$.
    - Then we can perform k insertions (each taking $O(\log n)$ time) to insert each node from H_2 into H_1.
    - Thus this takes $O(k \log n)$ in total.

Solution We present three ideas:

1. An algorithm works for tree-based implementation using worst case $O((\log n)^3)$ time.
    a. *Remark.* This algorithm can be improved to $O(\log k \cdot \log n))$.
2. A randomized algorithm (destroys *structural* property of heaps) where $T^{\text{exp}} = O(\log n)$.
3. An algorithm using higher-degree heaps with amortized runtime $O(\log n)$.

4.1 Worst-Case Heap Joining

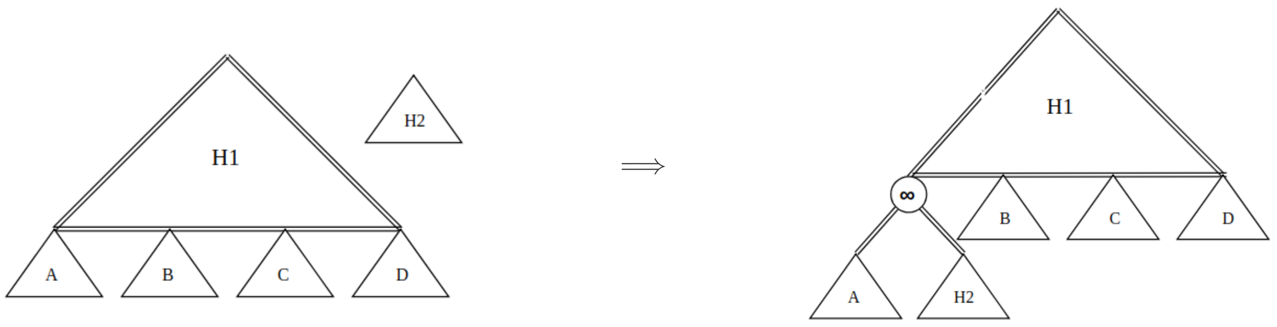
Given H_1, H_2, $|H_1| = n$, $|H_2| = k \leq n$, we want to create a heap combining both heaps.

Case I. Both heaps are full and they have the same height:

1. Create a new heap and set the root r's priority to ∞.
2. Set H_1 to be r's left child and H_2 to be r's right child.
3. Call `deleteMax()`. Since `deleteMax()` also takes care of structural property, we are done.

The runtime is dominated by `deleteMax()` which has $O(\log n)$ time.

Case II. Both heaps are full and H_2 has a smaller height:



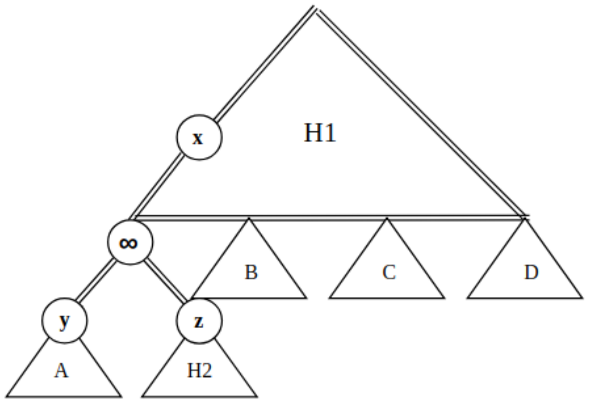
Phase I: Merge

0. Let $h(H)$ represent H's height. Since $h(H_2) < h(H_1)$, we can find a subheap of H_1 with the same height as H_2. In the left diagram, subheaps A, B, C, and D are such heaps.
1. Since $h(A) = h(H_2)$, we use **Case I**'s strategy to merge H_2 into A:
   a. Create a new heap and set the root r's priority to ∞.
   b. Set A and H_2 to be r's left and right child, respectively.
   c. Replace A with this new heap.

   We arrive at the right diagram.

Phase II: Adjust

To get rid of ∞ node, consider the following three nodes:



1. Node y: root of subheap A.
2. Node z: root of subheap H_2.
3. Node x: ∞'s parent, which originally was y's parent.

We have the following observations:

1. A and H_2 are valid heaps by construction, hence both satisfy the heap property.
2. By heap ordering property of H_1, x was y's parent previously implies $x > y$, so subheap A (and its root y) will not violate the heap property.
3. The only problem is we don't know about the relationship between x and z.
   a. If $x > z$, then we can bubble up ∞ and call `deleteMax()`.
   b. If $x < z$ however, the heap ordering property is broken. Moreover, if $x < z$, then there might exist more elements in H_2 that has a larger value than x, or more ancestors of x is less than z.

To fix the problem (when $x < z$):

1. For all ancestors a of x, call `fix-down(a)`.
   a. This guarantees all ancestors of x will be bubbled into the correct place.
   b. In the worst case, there are $\log n$ such ancestors ($h(H_1) = n$), and each `fix-down(a)` requires $\log n$ comparisons (again, $h(H_1) = n$), so overall this step takes $O(\log^2 n)$.
2. Call `deleteMax()`: After those `fix-down`'s, ∞ will be at the root position. Calling `deleteMax()` effectively removes ∞.

Overall, the runtime is dominated by those `fix-down`'s and hence has runtime $O(\log^2 n)$.

Case III. H_2 is full but H_1 is not.

This takes $O(\log^2 n)$ time. We omit this case.

Case IV. Both H_1, H_2 are not full.

1. We split H_2 into $O(\log k)$ heaps that are full.

```
                      A
            /                    \
           B                      K
      /         \            /         \
     C           D          k           K
   /   \       /   \      /   \       /   \
  C     C     E     F    K     K     K     K
 / \   / \   /
C   C C   C G
```

   a. In the above diagram, elements labeled with same letter are grouped into one full heap, e.g., the entire right subheap (labeled `k`) is a full heap, and the node `F` itself is a heap (because we have nothing to group it with).
   b. Since the heap is not full, there will always be a path from root to leaf, where each node on the path is a one-node heap. In the diagram above, this path is represented by `A -> B -> D -> E -> F`.
   c. Since the path has at least $\log k$ one-node heaps, overall we get $O(\log k)$ full heaps.
2. For each full subheap, merge them into H_1 using **Case III**. Since each merge takes $O(\log^2 n)$ and we have $O(\log k)$ full heaps, we end up with $O(\log k \cdot \log^2 n)$ time.

Conclusion Overall, the runtime is dominated by **Case IV**: $O(\log^3 n)$ time. $\square$

4.2 Expected $O(\log n)$ Merge

4.2.1 Pseudocode

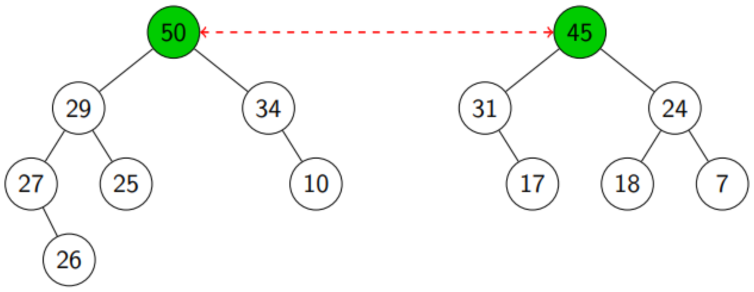
```
heapMerge(r1, r2)
r1, r2: roots of two heaps, possibly NIL
return: root of merged heap

1.  if r1 is NIL return r2
2.  if r2 is NIL return r1
3.  if key(r2) > key(r1)
4.    randomly pick one child c of r1
5.    replace subheap at c by heapMerge(c, r2)
6.    return r1
7.  else
8.    randomly pick one child d of r2
9.    replace subheap at d by heapMerge(r1, d)
10.   return r2
```

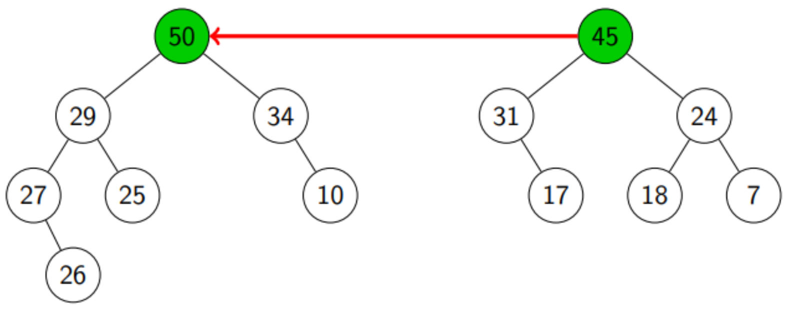
- Assume heaps are stored as binary trees but with heap ordering property.
- We merge the heap with smaller root into the other one.
- We randomly choose the child at which to merge, then recurse.

4.2.2 Graphical Illustration

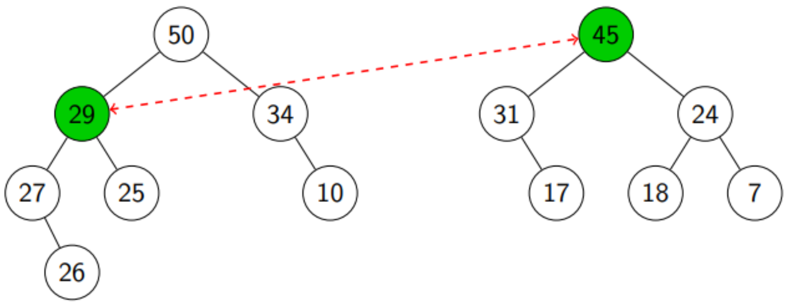
Suppose we want to merge the following two heaps.



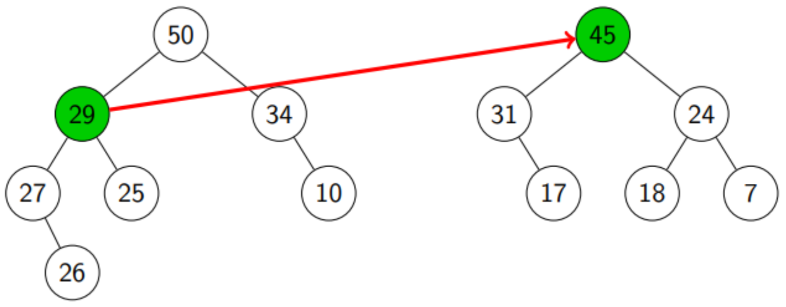
We see that root of left heap is greater than root of right heap (see line 7).



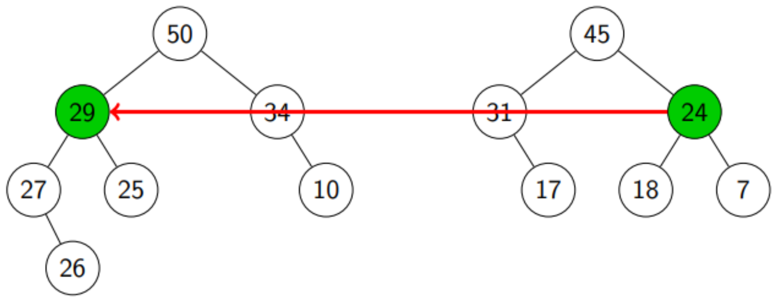
We randomly pick a child of 50 and ended up with 29 (see line 8).



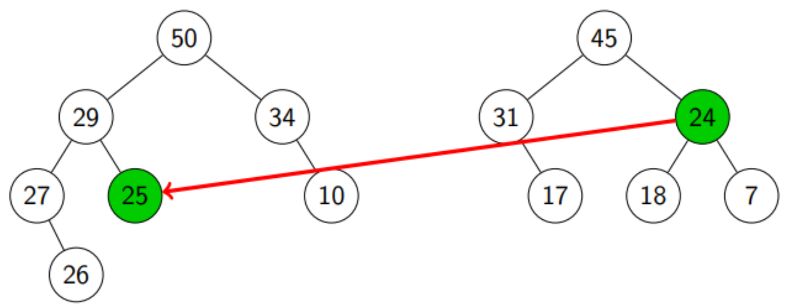
We see $29 < 45$ (see line 3).

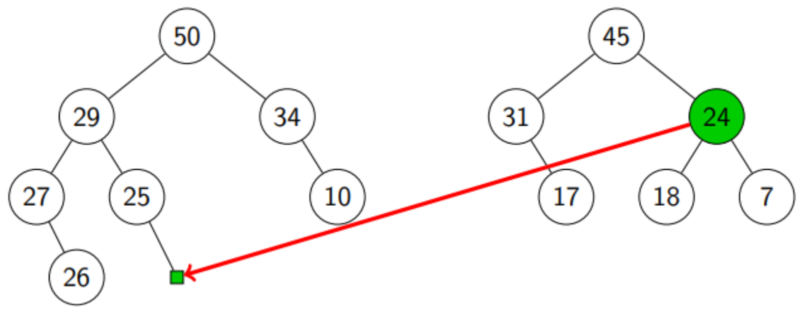


We randomly pick a child of 50 and ended up with 29 (see line 4).

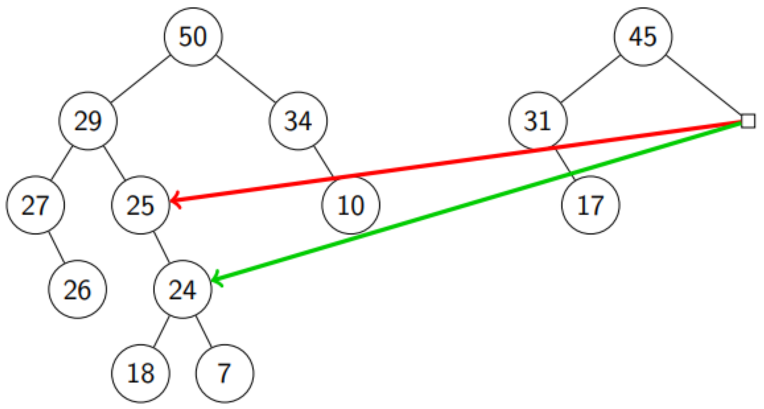


We keep doing this:

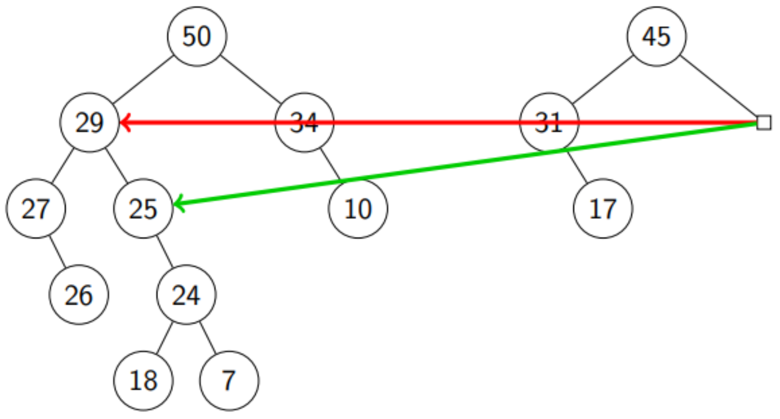




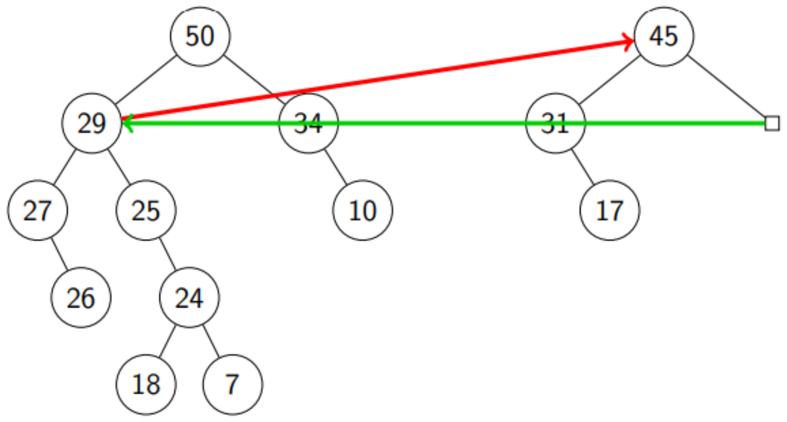
Finally, we see that `r1` is `NIL` and move r_2 (subheap with root 24).



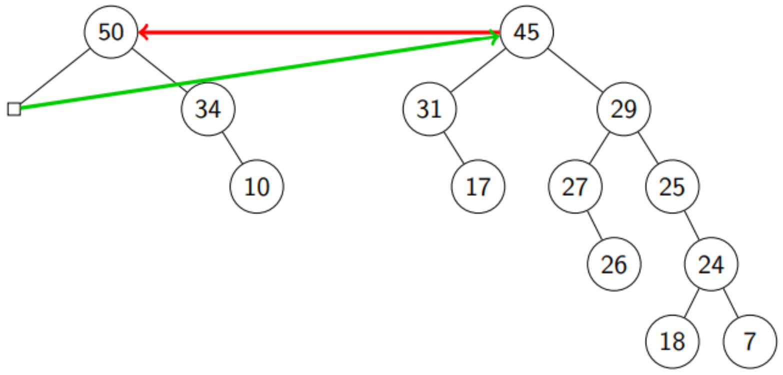
Return from call stack:



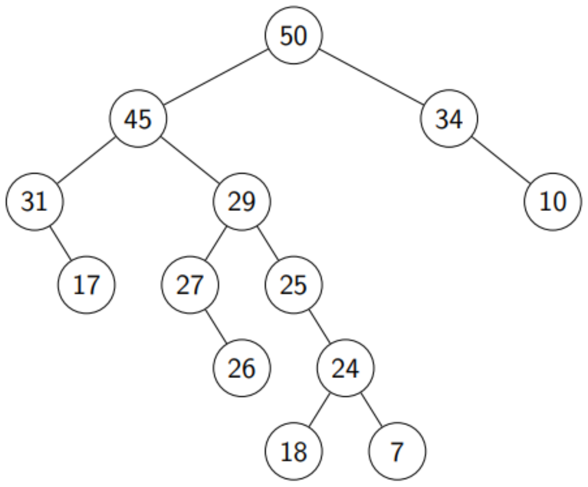
Return from call stack & compare 29 with 45: $29 < 45 \implies$ we move the subheap with root 29.



Return from call stack & compare 45 with 50: $50 > 45 \implies$ we move the subheap with root 50.



We are done!



4.2.3 Expected Runtime Analysis

Recall the expected runtime of a randomized algorithm can be expressed as:

$$\begin{aligned} T^{\exp}(I) &= \mathbb{E}[\text{runtime for } I] \\ &= \sum_{\text{sequences of random numbers}} \mathrm{P}(r \text{ is chosen}) \cdot (\text{runtime of } I \text{ when using sequence } r) \end{aligned}$$

$$T^{\exp}(n) = \max_{\text{instances } I \text{ of size } n} T^{\exp}(I)$$

We want to design the randomization such that *all instances of size n have equally good performance* because the expected runtime depends on the max of every instance.

Before we analyze the expected runtime of heap merge, let us first consider a simpler problem:

Problem Given a binary tree, you do a random walk downtime until you reach `NIL`. What is the runtime for this? We make no assumption on the tree, i.e., it may be very unbalanced.

Lemma The expected runtime $T^{\exp}(n)$ (i.e., on a tree of n nodes) is $O(\log n)$.

Proof. Assume that the time to go from a node to a randomly chosen child is at most c. We claim that $T(n) \leq c \cdot \log(n+1)$. We prove by induction.

- Base: $n = 1$.
  - LHS: $T(1) \leq c$ because we are one step away from `NIL`.
  - RHS: $c \log(2) = c$.
- Step: $n > 1$.
  - Let n_L, n_R represent the number of nodes in the left and right subtrees, respectively. Then $n_L + n_R = n - 1 \implies (n_L + 1) + (n_R + 1) = n + 1$.
  - The total runtime can be expressed s:

$$T(n) = c + \frac{1}{2}T(n_L) + \frac{1}{2}T(n_R) = c + \frac{1}{2}c\log(n_L+1) + \frac{1}{2}c\log(n_R+1)$$

    - c: time for choosing child and walking down
    - $\frac{1}{2}T(n_L)$: probability of choosing the left subtree times the runtime when left subtree is chosen.
    - $\frac{1}{2}T(n_R)$: probability of choosing the right subtree times the runtime when right subtree is chosen.
  - Recall concave functions has the property that

$$\forall x, y : \frac{1}{2}(f(x) + f(y)) \leq f\left(\frac{x+y}{2}\right)$$

    Thus,

$$\begin{aligned}
T(n) &= c + \frac{1}{2}T(n_L) + \frac{1}{2}T(n_R) \\
&= c + \frac{1}{2}c\log(n_L+1) + \frac{1}{2}c\log(n_R+1) \\
&= c + c\cdot\frac{1}{2}\big((\log n_L + 1) + (\log n_R + 1)\big) \\
&\le c + c\cdot\log\left(\frac{n_L+1+n_R+1}{2}\right) \\
&= c + c\cdot\big(\log(n+1) - \log 2\big) \\
&= c + c\cdot\log(n+1) - c \\
&= c\log(n+1) \qquad \square
\end{aligned}$$

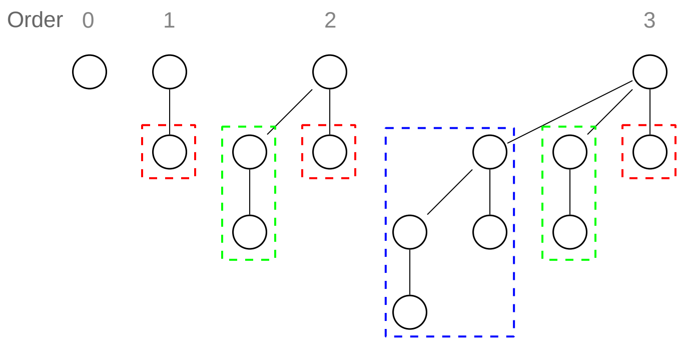
Back to the heap merge analysis, we can view it as going down in both heaps randomly until one hits a `NIL` child. Then

$$T^{\text{exp}}(n) \le 2\cdot \text{random walk in a tree with size } n \in O(\log n) \qquad \square$$

4.3 Binomial Heap

Recall that merge is easy for list ($O(1)$ for concatenation). To achieve our goal of worst-case $O(\log n)$ merging, we can store *a list of trees* that satisfy the heap-property.

4.3.1 Binomial Tree



A **binomial tree** is defined recursively as follows:

1. A binomial tree of order 0 is a single node.
2. A binomial tree of order k has a root node whose children are roots of binomial trees of order $k-1, k-2, \ldots, 2, 1, 0$ (in this order).

Because of its unique structure, a binomial tree of order k can be constructed from two trees of order $k-1$ trivially by attaching one of them as the left-most child of the root of the other tree. This feature is central to the merge operation of a binomial heap, which is its major advantage over the conventional heaps.

Remark. The name comes from the shape: a binomial tree of order n has $\binom{n}{d}$ nodes at depth d.

Claim. A binomial tree of order k has 2^k nodes with height k.

Proof. We prove by induction. The base case is trivial. Suppose this is true for $k-1$. Then binomial tree B_k contains two copies of B_{k-1}, so B_k has $2^{k-1}+2^{k-1}=2^k$ nodes. Because the max depth of a node in B_k is one greater than the max depth in B_{k-1}, by the inductive hypothesis, the maximum depth is $(k-1+1)=k$.

4.3.2 Binomial Heap

A **binomial heap** is implemented as a set (or list, as presented in our class) of binomial trees that satisfy the *binomial heap properties*:

1. Each binomial tree in a heap obeys the *heap ordering property.*
2. There can only be either *one* or *zero* binomial trees for each order, include zero order.

The first property ensures that the root of each binomial tree contains the maximum in the tree, which applies to the entire heap.

The second property implies that a binomial heap with n nodes consists of at most $1+\log n$ binomial trees. In fact, the number of orders of these trees are uniquely determined by the number of nodes n: eaach binomial tree corresponds to the 1 bits in the binary representation of number n. For example, if a heap contains $n=13=1101_2=2^3+2^2+2^0$ nodes, then this binomial heap with 13 nodes will consist of three binomial trees of order 3, 2, and 0.

4.3.3 Binomial Heap Operations

Let R be the list of trees and d_T be the degree at the root of tree T (take the max).

Merge: $O(1)$ as we are basically concatenating lists.

Insert: $O(1)$ for the same reason. In fact, insertion can be seen as merging as well.

DeleteMax: $O(|R|+\max_T d_T)$.

1. Locate the max: search among all roots, $|R|$ of them so $O(|R|)$ for searching.
2. Remove the max and merge the subtrees into the heap: suppose the tree has d_T children, then we need to perform d_T insertions. Hence $O(d_T)$ time for merging.
3. Cleanup: we want to merge trees together so after this step, no two trees have the same degrees in the list.

```
binomialHeapCleanup(R, n)
R: stack of trees (of a binomial heap of size n)
```

```
C <- array of length (log n + 1), initialized NIL
while R is non-empty:
  T <- R.pop() // extract a tree to work with
  d_T <- degree of root of T // check how many children root of T has
  if C[d_T] is NIL // if we haven't found another tree of the same degree
    C[d_T] <- T    // we put a pointer to T in the C[d_T]
  else // not NIL means we have saw another tree of the same degree
    T' <- C[d_T];  C[d_T] <- NIL // extract the previous result
    if key(root(T)) > key(root(T'))
      make root(T') a child of root(T)
      R.push(T)
    else
      make root(T) a child of root(T')
      R.push(T')
for i<-0 to log n: // Finally, we merge the binomial trees together
  if C[i] != NIL
    R.push(C[i])
```

4.3.4 Amortized Analysis (via Potential Function)

Lemma. If all trees were built with these operations, $d_T \leq \log n + 1$.

Proof. root degree $d \implies n = 2^d \implies d = \log n$. This also implies $\max_T d_T = O(\log n)$. $\square$

Define $\phi(t) = N_t \cdot c$, where N_t is the number of trees at time stamp t and c is a constant bigger than all constants in all other O terms. Recall the amortized runtime of step $t \to t+1$ is defined to be $\text{actual runtime} + \phi(t+1) - \phi(t)$.

To show `merge(R1, R2)` has amortized time $O(1)$:

- Actual runtime: $O(1) \implies c$.
- Potential difference: $N_{t+1} - N_t$, or the difference between number of trees at $t+1$ vs t. Before the merge there are $|R_1| + |R_2|$ trees. Since we are not modifying anything (only moving them), there are still the same number of trees.
- Putting them together: $T_{\text{amortized}} \leq c + (|R_1| + |R_2|) - (|R_1| + |R_2|) = c \in O(1)$.

Time to analyze `deleteMax()`:

- Actual time: $O(|R| + \max_T d_T) \implies c \cdot (|R| + \max_T d_T)$.
- Potential difference: $c|R'| - c|R|$, number of trees after minus number of trees before.
- Putting them together:

$$
\begin{aligned}
T_{\text{amortized}} &\le c \cdot (|R| + \max_T d_T) + c|R'| - c|R| \\
&= c \max_T d_T + c|R'| \\
&= c \max_T d_T + c \max_T d_T \\
&= 2c \max_T d_T \le 2c \log n \in O(\log n) \qquad \square
\end{aligned}
$$

where the $\max_T$ finds the max degree of all trees that could be in a binomial heap with n items.

4.3.5 Improvement on Worst-Case Performance

So far we have achieved $O(\log n)$ amortized runtime for `deleteMax()`. If we are willing to trade runtime for `merge`/`insert` for `deleteMax()`, we can achieve worst-case $O(\log n)$ for all three operations.

By calling `clean-up` after every operation (`insert`/`merge`), we can keep $|R| \in O(\log n)$. Recall `clean-up` takes $O(|R| + \max_T d_T)$. Enforcing this would make `insert`/`merge`/`deleteMax` all achieve $O(\log n)$ worst-case runtime.

4.3.6 Magic Trick

We can use a binary tree T' to store a tree T with an arbitrary degree!

The new binary tree T' has the same nodes as T. For node v in T', its left child in T' is the left-most child in T and right child in T' is its sibling in T.

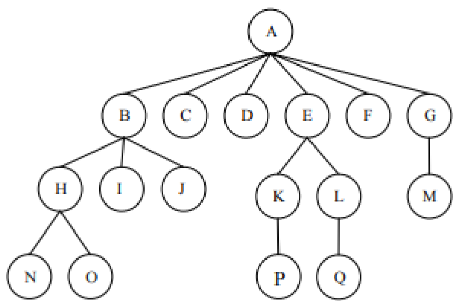


Figure 1. A sample *k*-ary tree, where k = 6.

$\Longrightarrow$

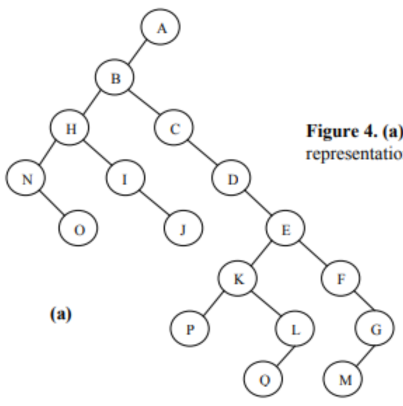


Figure 4. (a) The equivalent binary tree representation of the given *k*-ary tree in Figure 1.

Source: A very very interesting paper to read.

V1.0: Feb. 3, David