Greedy Algorithms Overview

Last Updated: October 23rd, 2023

1 Introduction

A **greedy algorithm** is often considered the easiest of algorithms to describe and implement, and is characterized by the following two properties:

1. the algorithm works in successive stages, and during each stage a choice is made that is locally optimal
2. the sum totality of all the locally optimal choices produces a globally optimal solution

If a greedy algorithm does not always lead to a globally optimal solution, then we refer to it as a **heuristic**, or a **greedy heuristic**. Heuristics often provide a "short cut" (not necessarily optimal) solution.

The following are some computational problems that that can be solved using a greedy algorithm.

Huffman Coding finding a code for a set of items that minimizes the expected code-length

Minimum Spanning Tree finding a spanning tree for a graph whose weighted edges sum to a minimum value

Single source distances in a graph finding the distance from a source vertex in a weighted graph to every other vertex in the graph

Fractional Knapsack selecting a subset of items to load in a container in order to maximize profit

Task Selection finding a maximum set of timewise non-overlapping tasks (each with a fixed start and finish time) that can be completed by a single processor

Unit Task Scheduling with Deadlines finding a task-completion schedule for a single processor in order to maximize the total earned profit

Like all families of algorithms, greedy algorithms tend to follow a similar analysis pattern.

Greedy Correctness Correctness is usually proved through some form of induction. For example, assume their is an optimal solution that agrees with the first k choices of the algorithm. Then show that there is an optimal solution that agrees with the first $k + 1$ choices. Conclude that the greedy solution is in fact optimal.

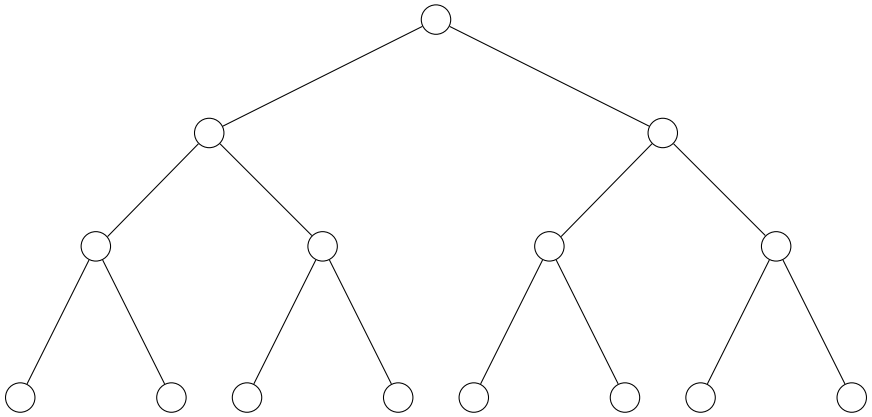
Greedy Complexity The running time of a greedy algorithm is determined by the ease in maintaining an ordering of the candidate choices in each round. This is usually accomplished via a static or dynamic sorting of the candidate choices.

Greedy Implementation Greedy algorithms are usually implemented with the help of a static sorting algorithm, such as Quicksort, or with a dynamic sorting structure, such as a binary heap. Additional data structures may be needed to efficiently update the candidate choices during each round.

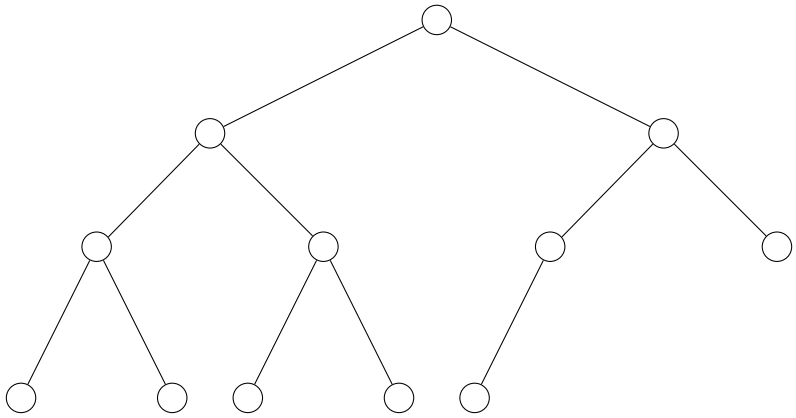
2 Data Structures for Greedy Algorithms

2.1 Binary Heaps

Definition 2.1. A **perfect binary tree** is a binary tree whose leaves all have the same depth and an **almost-perfect binary tree** is a perfect binary tree that is missing zero or more leaves (starting from the far right and moving left) at the last level. Thus, a perfect binary tree is a special case of an almost-perfect binary tree.



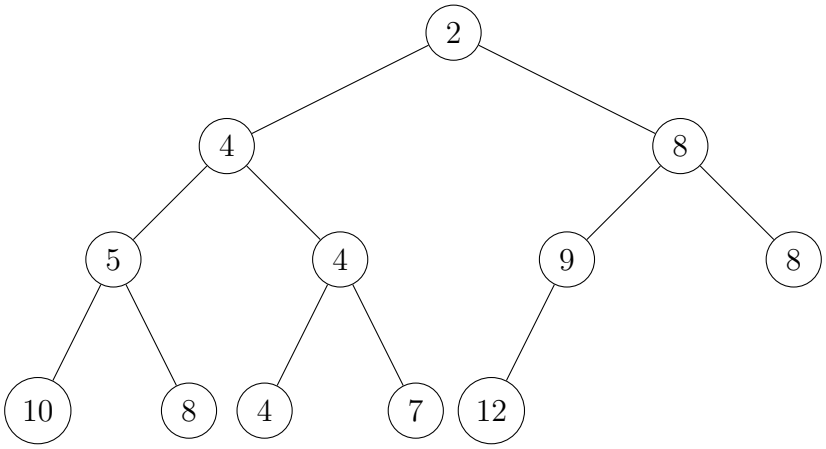
Example of a perfect tree.



Example of an almost-perfect binary tree.

Definition 2.2. A **binary min heap** H may be defined as an almost-perfect binary tree, where each tree node n stores a member s_n of an ordered set S, and has the property that if n' is a child of n, then $s_n \leq s_{n'}$. For $s_1, s_2 \in S$, if

1. $s_1 < s_2$, then s_1 has **higher priority** than s_2.
2. $s_1 > s_2$, then s_1 has **lower priority** than s_2.
3. $s_1 = s_2$, then s_1 and s_2 **equal priority**.



Example of a binary min heap whose nodes store integers.

Note that a heap may be implemented as an array, where

1. the root is stored at index 1
2. if node n is stored at index i, then its left (respectively, right) child is stored at index $2i$ (respectively, $2i + 1$).

Proposition 2.3. A binary heap H of size n (nodes) has height equal to $\lfloor \log n \rfloor$.

Proof. If H is a perfect tree, then it has $n = 2^{h+1} - 1$ nodes, where h is the tree height. Thus,

$$h = \lfloor \log(2^{h+1} - 1) \rfloor = \lfloor \log n \rfloor.$$

If H is an almost-perfect tree with at least one missing node, then it has $n = 2^h + k$ nodes, where $0 \leq k < 2^h$, and

$$h = \lfloor \log(2^h + k) \rfloor = \lfloor \log n \rfloor,$$

since

$$2^h \leq 2^h + k < 2^h + 2^h = 2 \cdot 2^h = 2^{h+1}. \quad \square$$

2.2 Heap Operations

void insert(Item i) insert an item into the heap.

- **Percolate up:** insert i at the end of the bottom level and, while its parent has a lower priority, swap i with its parent.
- $O(\log n)$ complexity

Item pop() Remove from the heap item i stored at the root and return i.

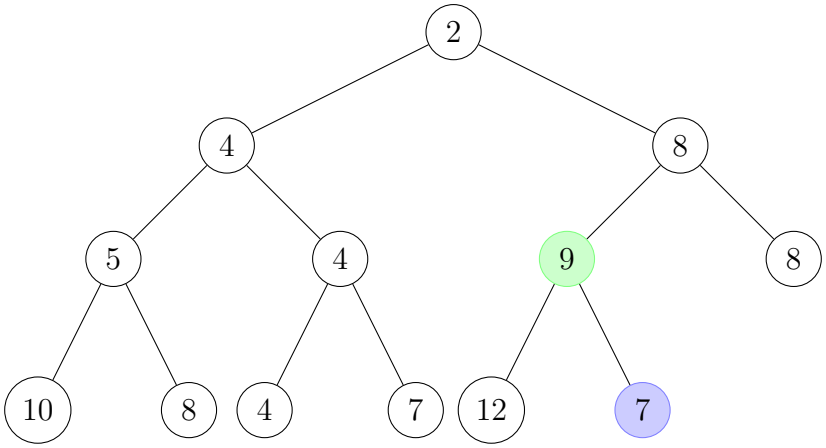
- **Percolate down:** replace root node with the last node n in the bottom level and, while n's highest-priority child c has higher priority than n, swap n with c.
- $O(\log n)$ complexity

void increase_priority(Item i, Numeric δ) Increase i's priority by the amount of δ.

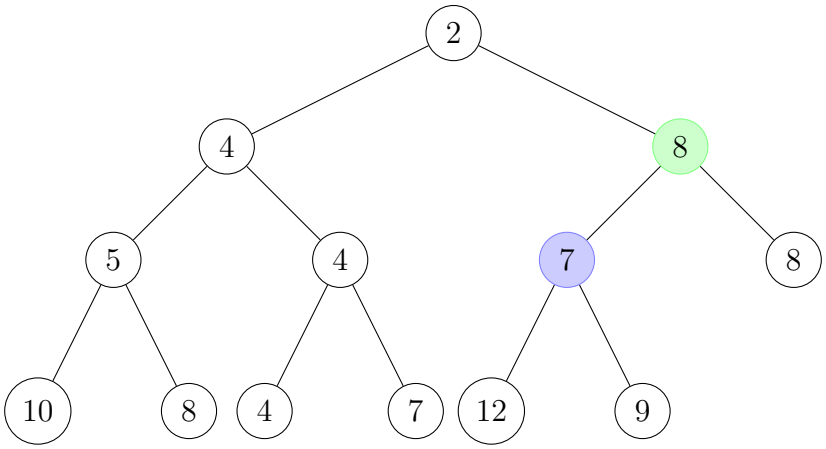
- Increase i's priority and then percolate it up the tree (see `insert()`) starting at its current location.
- $O(\log n)$ complexity

void build_heap(Item array[ )] build a heap from an array of items. Complexity: $\Theta(n)$

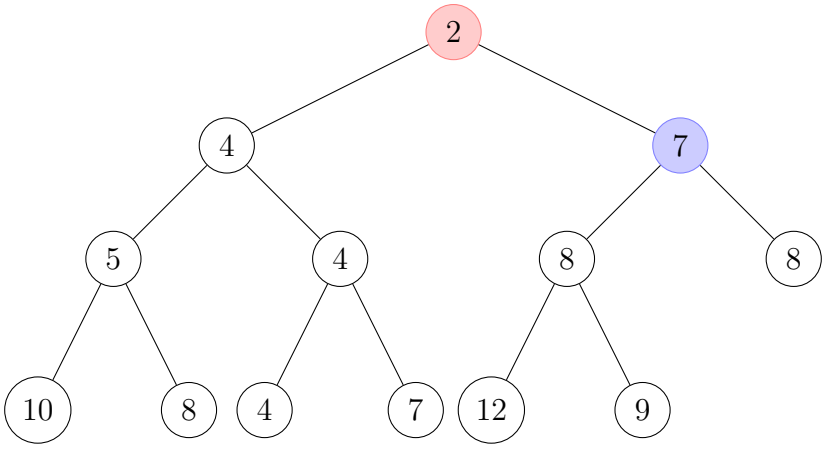
Example 2.4. The following is an example of inserting an item (7 in blue) into a binary min heap.



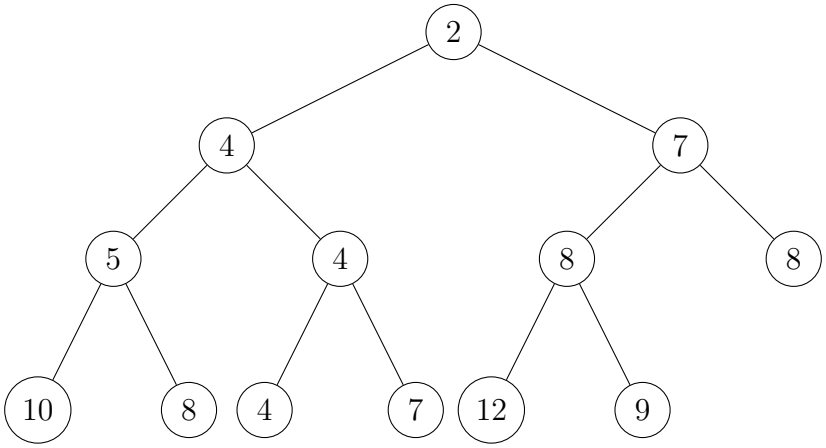
insert(): item 7 added to end of heap.



insert(): item 7 swapped with parent 9.

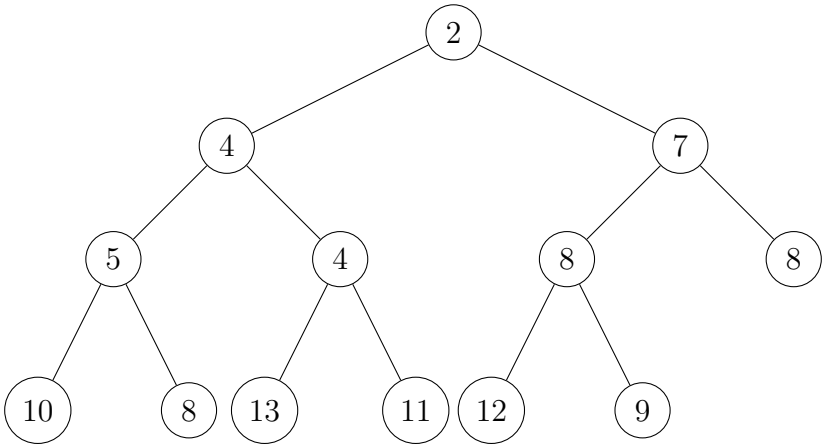


insert(): item 7 swapped with parent 8.

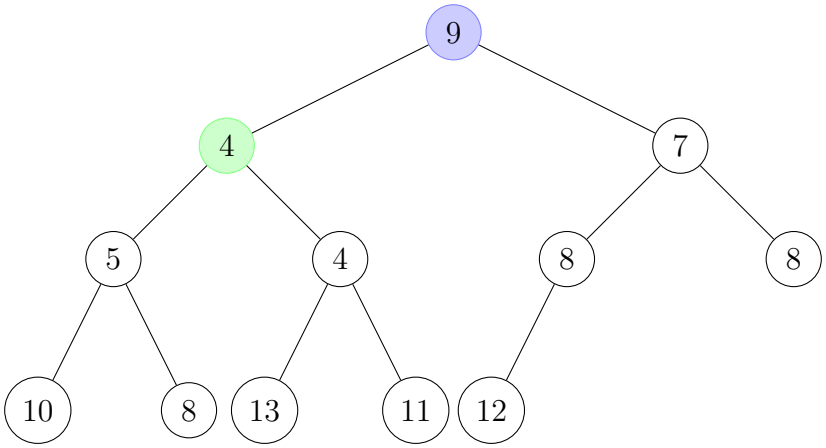


insert(): final state.

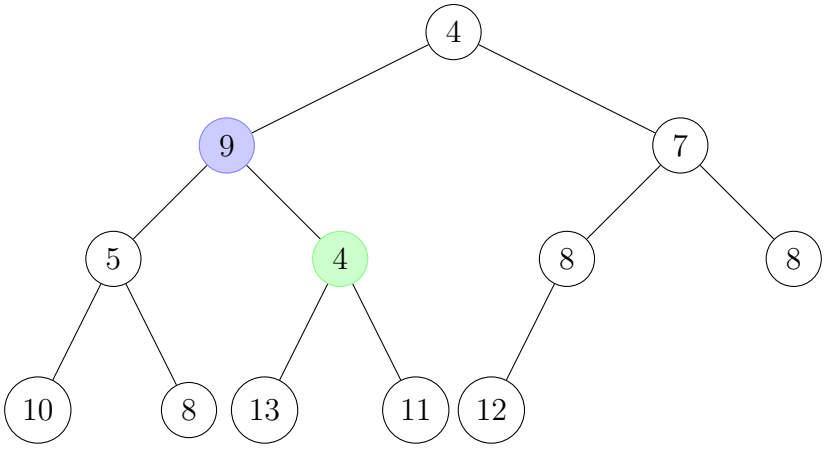
Example 2.5. The following is an example of popping a binary min heap.



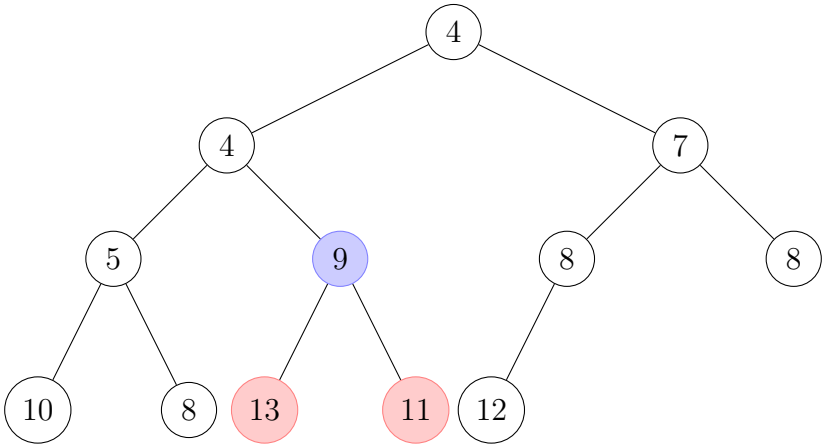
pop(): initial state.



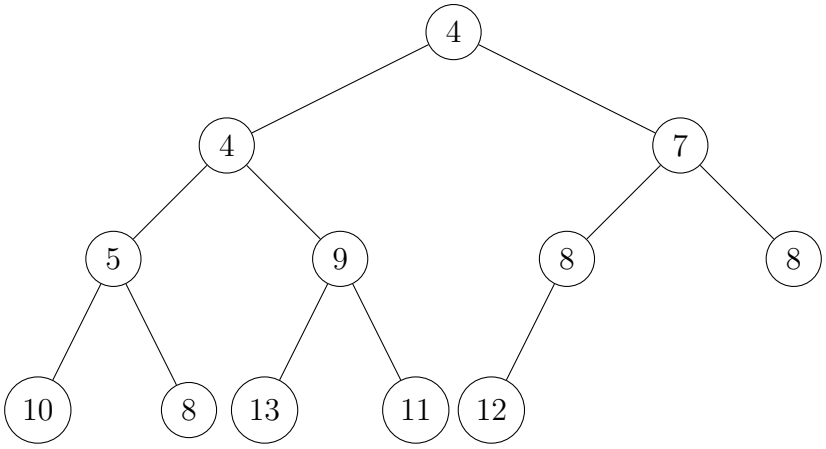
pop(): root node 2 removed and replaced by end-of-heap node 9.



pop(): item 9 swapped with item 4.

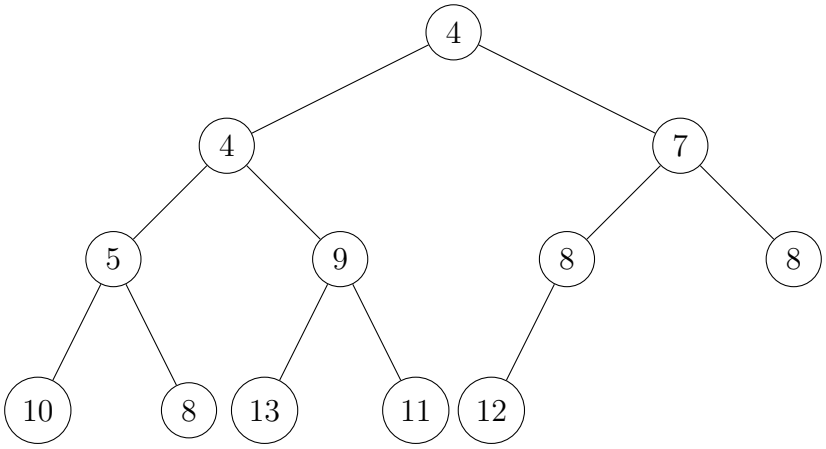


pop(): item 9 swapped with item 4.

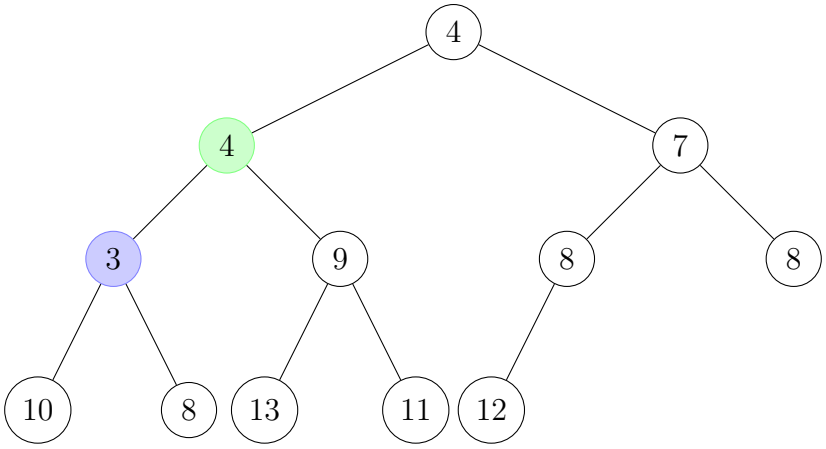


pop(): final state.

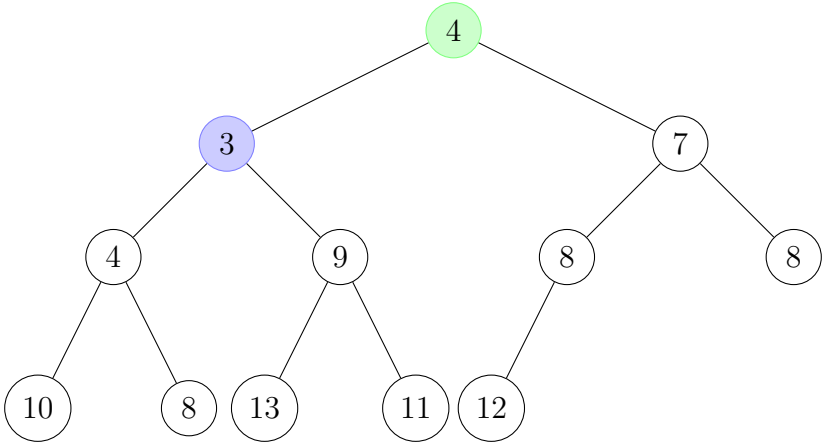
Example 2.6. The following is an example of increasing a node's priority.



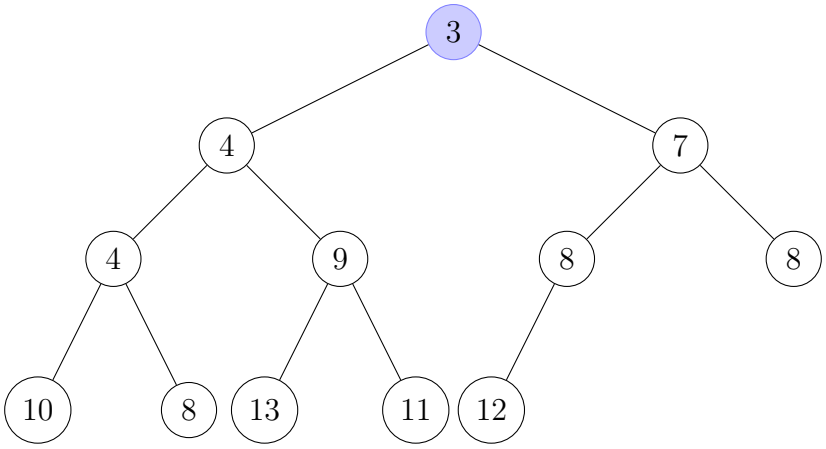
increase_priority: initial state.



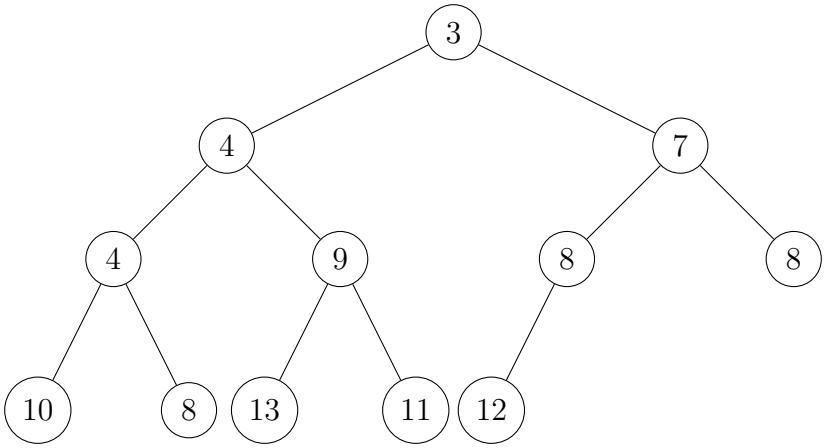
increase_priority: increase 5's priority to 3.



increase_priority: swap 3 with parent 4.



increase_priority: swap 3 with parent 4.



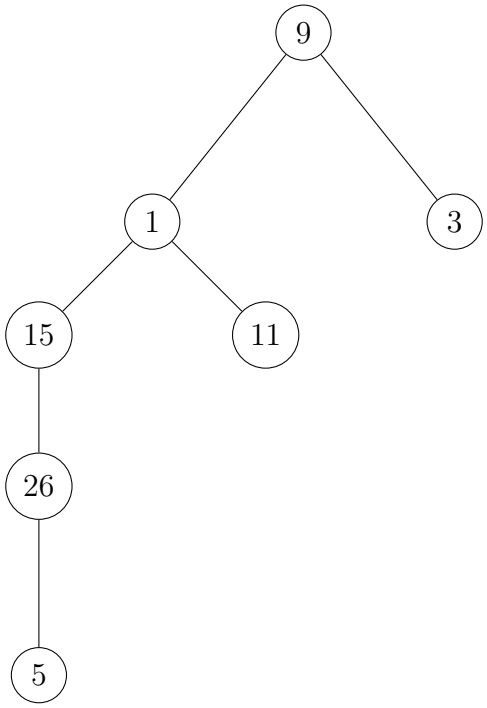
increase_priority: final state.

2.3 Disjoint Sets Data Structure

The **disjoint sets data structure** is used for maintaining a collection of disjoint sets of some underlying set. In what follows we describe how each set in the collection can be represented by a **membership tree (M-tree)**, where each M-tree can be represented by a collection of nodes, with each being called a **membership node (M-node)**.

```
structure MNode
{
    MNode parent; //reference to the parent of this MNode
    Item item; //the item being stored in this MNode
}
```

Example 2.7. Suppose the underlying set is the set of nonnegative integers, and consider the subset $S = \{1, 3, 5, 9, 11, 15, 26\}$. This set can be repesented by a the following tree.



A tree representation of $S = \{1, 3, 5, 9, 11, 15, 26\}$.

Notice that the order of the numbers does not matter. All that matters is that

1. each member of S is stored in exactly one of the M-nodes, and
2. every item stored in an M-node is a member of S.

Notice also that the parent of 5 is 26, the parent of 26 is 15, etc..

2.4 Disjoint Set Operations

void union(MNode n_1, MNode n_2) has the effect of setting n_2's parent to n_1.

- O(1) complexity since it requires a single assignment.

MNode root(MNode n) Returns the root node r of the tree in which n is located.

- **Path Compression:** has the side effect of setting n''s parent to r, for every n' along the path from n to r (except for r).
- Complexity is proportional to the path length from n to r.

Theorem 2.8. If one begins with n disjoint singleton sets and performs a sequence of m `union` and `root` operations (without necessarily using path compression), then the total running time is equal to $O(n + m \log n)$. Note: this result assumes that, when performing unions, the smaller tree is merged into the larger one.

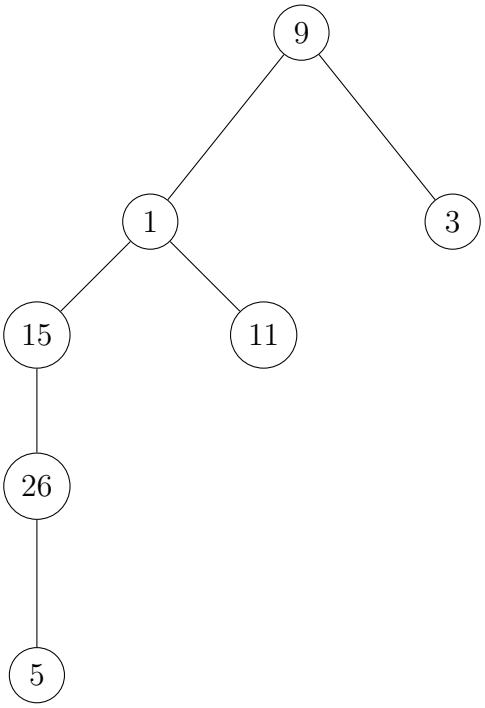
Theorem 2.9. If one begins with n disjoint singleton sets and performs a sequence of m `union` and `root` operations using path compression and merging smaller into larger trees, then the total running time is equal to $O(n + m\alpha(n))$, where $\alpha(n) = o(\log^*(n))$, where $\log^* n$ equals the number of iterative applications of the log function to n before the result is less than or equal to 1. For example, $\log^*(2^{64}) = 5$.

Example 2.10. Suppose that we begin with the singleton sets

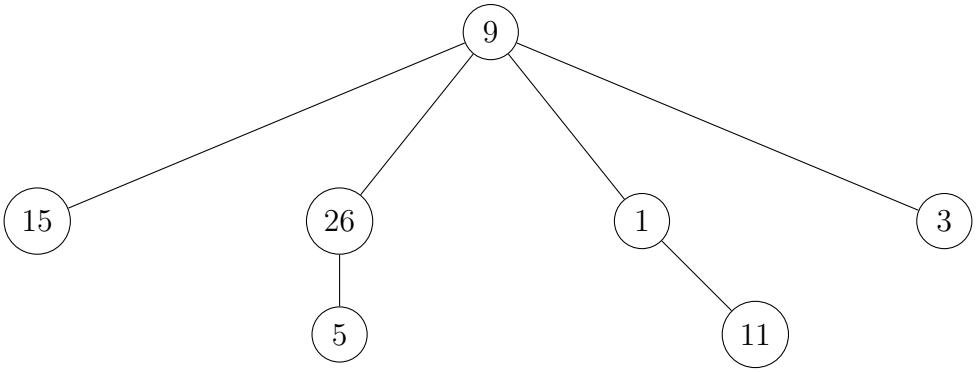
$$\{1\}, \{3\}, \{5\}, \{9\}, \{11\}, \{15\}, \{26\}.$$

verify that the `union` operations, `union`$(1, 11)$, `union`$(26, 5)$, `union`$(9, 3)$, `union`$(15, 26)$, `union`$(1, 15)$, `union`$(9, 1)$ results in the tree shown in Example 2.7 (assuming the children of each node are unordered).

Example 2.11. The following is an example of path compression as a side effect of the operation `root`(26).



The tree for which 26 is a member.



The resulting tree after `root`(26) has been executed.

2.5 Application: Unit Task Scheduling with Deadlines

The input for this problem is a set of n tasks $a_1, \ldots, a_n$. The tasks are to be executed by a single processor starting at time $t = 0$. Each task a_i requires one unit of processing time, and has an integer deadline d_i. Moreover, if the processor finishes executing a_i at time t, where $d_i \leq t$, then a profit p_i is earned. For example, if task a_1 has a deadline of 3 and a profit of 10, then it must be either the first, second, or third task executed in order to earn the profit of 10. Consider the following greedy algorithm for maximizing the total profit earned. Sort the tasks in decreasing order of profit. Then for each task a_i in the ordering, schedule a_i at time $t \leq d_i$, where t is the latest time that does not exceed d_i, and for which no other task has yet to be scheduled at time t. If no such t exists, then skip a_i and proceed to the next task in the ordering.

Example 2.12. Apply the algorithm described above to the following problem instance. Note: if two tasks have the same profit, then ties are broken by alphabetical order. For example, Task b preceeds Task e in the ordering.

Task	a	b	c	d	e	f	g	h	i	j	k
Deadline	4	3	1	4	3	1	4	6	8	2	7
Profit	40	50	20	30	50	30	40	10	60	20	50

When executing the greedy algorithm for `UTS` described above, when inserting a task a at the time slot for when it will be executed, if one uses a naive approach that starts at a's deadline and linearly scans left until an open time slot is found, then the worst case occurs when each of the n tasks has a deadline equal to n and all tasks have the same profit. In this case task 1 is scheduled at n, task 2 at $n-1$, etc.. Notice that, when scheduling task i, the array A that stores the scheduled tasks at there chosen time slots must be accessed $i-1$ times before finding the available time $n-i+1$. This yields a total of

$$0 + 1 + \cdots + n - 1 = \Theta(n^2).$$

accesses. Thus, the algorithm has a running time of $T(n) = \mathrm{O}(n^2)$.

To improve the running time, we may associate an M-tree with each time slot associated with array A that provides the final scheduling of the tasks. Then if MNode n_i is associated with time slot (i.e. array index) i and belongs in M-Tree T, then any task with a deadline equal to i is scheduled at time j, where the MNode n_j is the root of T. Thus, scheduling a task requires a single `root` operation, followed by a single union operation for which the M-tree associdated with time j is merged with the M-tree associated with time $j-1$. This is necessary since time j is no longer available, and so any task that is directed to j must now be re-directed to a time for which any $(j-1)$-deadline task would get directed. Thus, a total of $2n$ root and `union` operations are required, yielding a running time of $T(n) = \mathrm{O}(n \log^* n)$. Therefore, the worst-case running time is the $\Theta(n \log n)$ time required to sort the tasks.

Example 2.13. Show the resulting disjoint-set data structure forest for the instance of `UTS` for which each of four tasks has a deadline equal to 3, and a profit equal to some constant P.

3 Exercises

1. Repeat Example 2.13, but now with the following insertions into an initially empty array: a at (array index) 3, b at 4, c at 4, d at 3, e at 4. Show the M-node connections after each insertion.

2. The **Fuel Reloading Problem** is the problem of traveling in a vehicle from one point to another, with the goal of minimizing the number of times needed to re-fuel. It is assumed that travel starts at point 0 (the origin) of a number line, and proceeds right to some final point $F > 0$. The input includes F, a list of stations $0 < s_1 < s_2 < \cdots < s_n < F$, and a distance d that the vehicle can travel on a full tank of fuel before having to re-fuel. Consider the greedy algorithm which first checks if F is within d units of the current location (either the start or the current station where the vehicle has just re-fueled). If F is within d units of this location, then no more stations are needed. Otherwise it chooses the next station on the trip as the furthest one that is within d units of the current location. Apply this algorithm to the problem instance $F = 25$, $d = 6$, and

$$s_1 = 4, s_2 = 7, s_3 = 11, s_4 = 13, s_5 = 18, s_6 = 20, s_7 = 23.$$

3. Given a finite set T of tasks, where each task t is endowed with a start time $s(t)$ and finish time $f(t)$, the goal is to find a subset T_{opt} of T of maximum size whose tasks are pairwise non-overlapping, meaning that no two tasks in T_{opt} share a common time in which both are being executed. This way a single processor can complete each task in T_{opt} without any conflicts.

   Consider the following greedy algorithm, called the **Task Selection Algorithm (TSA)**, for finding T_{opt}. Assume all tasks start at or after time 0. Initialize T_{opt} to the empty set, and initialize variable `last_finish` to 0. Repeat the following step. If no task in T has a start time equal to or exceeding `last_finish`, then terminate the algorithm and return T_{opt}. Otherwise add to T_{opt} the task $t \in T$ for which $s(t) \geq$ `last_finish` and whose finish time $f(t)$ is a minimum amongst all such tasks. Set `last_finish` to $f(t)$.

   Demonstrate TSA on the following set of tasks.

Task ID	Start time	Finish Time
1	2	4
2	1	4
3	2	7
4	4	8
5	4	9
6	6	8
7	5	10
8	7	9
9	7	10
10	8	11

4. Describe an efficient implementation of the Task Selection algorithm, and provide the algorithm running time under this implementation.

5. The **Fractional Knapsack** takes as input a set of goods G that are to be loaded into a container (i.e. knapsack). When good g is loaded into the knapsak, it contributes a weight of $w(g)$ and induces a profit of $p(g)$. However, it is possible to place only a fraction α of a good

into the knapsack. In doing so, the good contributes a weight of $\alpha w(g)$, and induces a profit of $\alpha p(g)$. Assuming the knapsack has a weight capacity $M \geq 0$, determine the fraction $f(g)$ of each good that should be loaded onto the knapsack in order to maximize the total container profit.

The Fractional Knapsack greedy algorithm (FKA) solves this problem by computing the **profit density** $d(g) = p(g)/w(g)$ for each good $g \in G$. Thus, $d(g)$ represents the profit per unit weight of g. FKA then sorts the goods in decreasing order of profit density, and initializes variable `RC` to M, and variable `TP` to 0. Here, `RC` stands for "remaining capacity", while `TP` stands for "total profit". Then for each good g in the ordering, if $w(g) \leq$ `RC`, then the entirety of g is placed into the knapsack, `RC` is decremented by $w(g)$, and `TP` is incremented by $p(g)$. Otherwise, let $\alpha =$ `RC`$/w(g)$. Then $\alpha w(g) =$ `RC` weight units of g is addded to the knapsack, `TP` is incremented by $\alpha p(g)$, and the algorithm terminates.

For the following instance of the FK problem, determine the amount of each good that is placed in the knapsack by FKA, and provide the total container profit. Assume $M = 10$.

good	**weight**	**profit**
1	3	4
2	5	6
3	5	5
4	1	3
5	4	5

6. Describe an efficient implementation of the FK algorithm, and provide the algorithm running time under this implementation.

7. The **0-1 Knapsack** problem is similar to Fractional Knapsack, except now, for each good $g \in G$, either all of g or none of g is placed in the knapsack. Consider the following modification of the Fractional Knapsack greedy algorithm. If the weight of the current good g exceeds the remaining capacity `RC`, then g is skipped and the algorithm continues to the next good in the ordering. Otherwise, it adds all of g to the knapsack and decrements `RC` by $w(g)$, while incrementing `TP` by $p(g)$. Verify that this modified algorithm does *not* produce an optimal knapsack for the problem instance of Exercise 5.

8. Given the set of keys $1, \ldots, n$, where key i has weight w_i, $i = 1, \ldots, n$. The weight of the key reflects how often the key is accessed, and thus heavy keys should be higher in the tree. The Optimal Binary Search Tree problem is to construct a binary-search tree for these keys, in such a way that

$$\text{wac}(T) = \sum_{i=1}^{n} w_i d_i$$

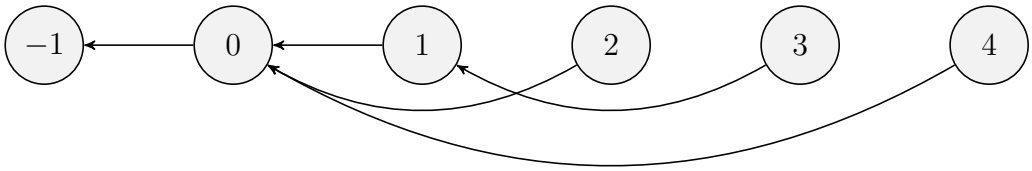
is minimized, where d_i is the depth of key i in the tree (note: here we assume the root has a depth equal to one). This sum is called the **weighted access cost**. Consider the greedy heuristic for Optimal Binary Search Tree: for keys $1, \ldots, n$, choose as root the node having the maximum weight. Then repeat this for both the resulting left and right subtrees. Apply this heuristic to keys 1-5 with respective weights 50,40,20,30,40. Show that the resulting tree does not yield the minimum weighted access cost.

9. Given a simple graph $G = (V, E)$, a **vertex cover** for G is a subset $C \subseteq V$ of vertices for which every edge $e \in E$ is incident with at least one vertex of C. Consider the greedy heuristic for

finding a vertex cover of minimum size. The heuristic chooses the next vertex to add to C as the one that has the highest degree. It then removes this vertex (and all edges incident with it) from G to form a new graph G'. The process repeats until the resulting graph has no more edges. Give an example that shows that this heuristic does not always find a minimum cover.

4 Exercise Solutions

1. Below is the final M-tree after all insertions have been made.



2. Minimal set of stations: s_1, s_2, s_4, s_5, s_7.

3. TSA returns $T_{\text{opt}} = \{1, 4, 10\}$.

4. It is sufficient to represent the problem size by the number n of input tasks. Sort the tasks in order of increasing start times. Now the algorithm can be completed in the following loop.

```
earliest_finish <- INFINITY
output <- EMPTY_SET

for each task t
     if f(t) < earliest_finish
          earliest_finish <- f(t)
          next_selected <- t

     else if s(t) >= earliest_finish
          earliest_finish <- f(t)
          output += next_selected
          next_selected <- t
output += next_selected
```

The above code appears to be a correct implementation of TSA. The only possible concern is for a task t that neither satisfies the `if` nor the `else-if` condition. Such tasks never get added to the final set of non-overlapping tasks. To see that this is justified, suppose in the `if` statement t is comparing its finish time $f(t)$ with that of t'. Then we have

$$s(t') \leq s(t) < f(t'),$$

where the first inequality is from the fact that the tasks are sorted by start times, and the second inequality is from the fact that t does not satisfy the `else-if` condition. Hence, it follows that t and t' overlap, so, if t' is added to the optimal set, then t should not be added. Moreover, the only way in which t' is not added is if there exists a task t'' that follows t in terms of start time, but has a finish time that is less than that of t''s. In this case we have $s(t) \leq s(t'')$ and $f(t) \geq f(t') \geq f(t'')$ and so t overlaps with t''. And once again t should not be added to the final set.

Based on the above code and analysis, it follows that TSA can be implemented with an initial sorting of the tasks, followed by a linear scan of the sorted tasks. Therefore, $T(n) = \Theta(n \log n)$.

5. The table below shows the order of each good in terms of profit density, how much of each good was placed in the knapsack, and the profit earned from the placement. The total profit earned is 14.4.

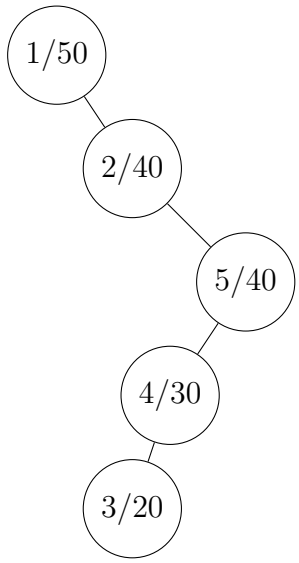
good	weight	profit	density	placed	profit earned
4	1	3	3	1	3
1	3	4	1.3	3	4
5	4	5	1.25	4	5
2	5	6	1.2	2	2.4
3	5	5	1	0	0

6. The parameters n, and $\log M$ can be used to represent the problem size, where n is the number of goods. Notice how $\log M$ is used instead of M, since $\log M$ bits are needed to represent capacity M. Furthermore, assume each good weight does not exceed M, and the good profits use a constant number of bits. Then the sorting of the goods requires $\Theta(n \log n)$ steps, while the profit density calculations and updates of variables `RC` and `TP` require $O(n \log M + n)$ total steps. Therefore, the running time of FKA is $T(n) = O(n \log n + n \log M)$.

7. The table below shows the order of each good in terms of profit density, how much of each good was placed in the knapsack by modified FKA, and the profit earned from the placement. The total profit earned is 12. However, placing goods 2, 4, and 5 into the knapsack earns a profit of $14 > 12$. An alternative algorithm for 0-1 Knapsack will be presented in the Dynamic Programming lecture.

good	weight	profit	density	placed	profit earned
4	1	3	3	1	3
1	3	4	1.3	3	4
5	4	5	1.25	4	5
2	5	6	1.2	2	0
3	5	5	1	0	0

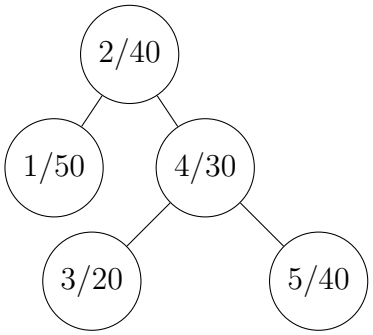
8. The heuristic produces the tree below.



Its weighted access cost equals

$$50(1) + 40(2) + 40(3) + 30(4) + 20(5) = 470.$$

However, a binary-search tree with less weighted-access cost (380) is shown below.



9. In the graph below, the heuristic will first choose vertex a, followed by four additional vertices (either b, d, f, h, or c, e, g, i), to yield a cover of size five. However, the optimal cover $\{c, e, g, i\}$ has a size of four.