CS580 Algorithm Design, Analysis, and Implementation

Lecture notes, Jan 25, 2022
Wufei Ma

1 Search Trees

Binary tree. Each node has two children, which could be empty. It also has a parent, which could also be empty.

- A node without any children (both empty) is a **leaf** node.
- The node without a parent is a **root** node.
- γ is a **descendant** of μ if γ is a child of μ or a descendant of a child of μ.
- If γ is a descendant of μ, then μ is an **ancestor** of γ.
- The **depth** of μ is the number of edges from root to μ.
- The **height** of μ is the max number of edges from μ to a leaf.
- The **subtree** of μ consists of μ and its descendant.

A node can be implemented with three node pointers and a key, and a tree can be represented by a node.

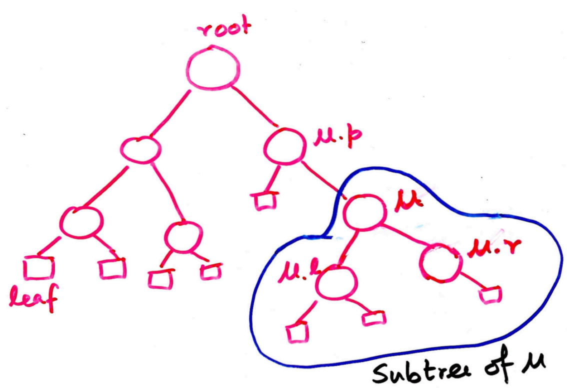

Figure 1: A binary tree.

Traversals. There are three types of traversals, in-order, pre-order, and post-order.

```
Procedure In-Order(u):
    if u != null then:
        In-Order(u.l)
        print(u.key)
        In-Order(u.r)
    endif
```

```
Procedure Pre-Order(u):
    if u != null then:
        print(u.key)
        In-Order(u.l)
        In-Order(u.r)
    endif
```

```
Procedure Post-Order(u):
    if u != null then:
        In-Order(u.l)
        In-Order(u.r)
        print(u.key)
    endif
```

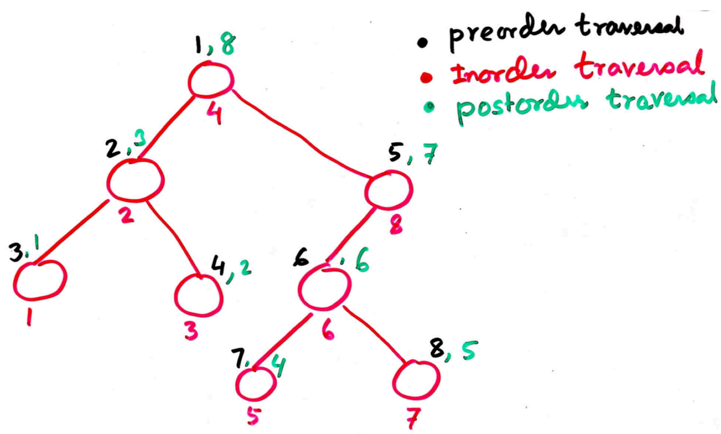

Figure 2: Tree traversals.

A property. μ is an ancestor of γ if and only if $\text{pre}(\mu) < \text{pre}(\gamma)$ and $\text{post}(\mu) > \text{post}(\gamma)$.

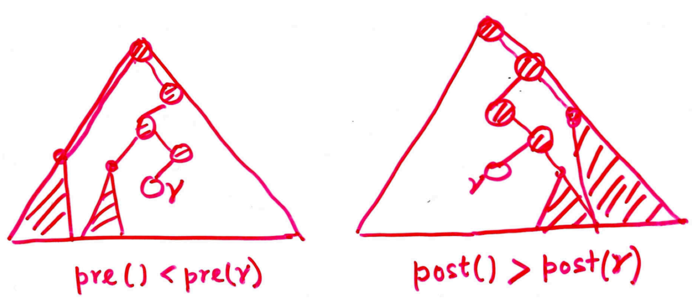

Figure 3: A property.

Binary search tree. A binary tree is a **binary search tree** if the keys printed in-order is sorted.

All operations on binary search tree, including `Search`, `Min`, `Successor`, and `Insertion`, are in $O(h)$.

Searching.

```
Function Search(x, u):
    if u == null or x = u.key then:
        return u
    else:
        if x < u.key then:
            return Search(x, u.l)
        else:
            return Search(x, u.r)
```

Insertion.

```
Procedure Insert(p, r):
    # insert r into a tree denoted by p
    x = null
    u = p
    while u != null:
        x = u
        if r.key < u.key then:
            u = u.l
        else:
            u = u.r
        endif
    endwhile
    r.p = x
    if x == null then:
        p = r
    else:
        if r.key < x.key then:
            x.l = r
        else:
            x.r = r
        endif
    endif
```

Deletion.

1. **Case I:** The node has no children. Just delete this node.
2. **Case II:** The node has one child. Replace with the child.
3. **Case III:** The node has two children. Replace the key with its successor. Delete its successor. (The successor has at most one child).

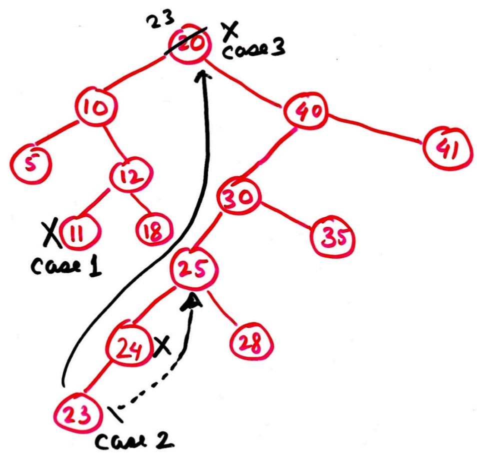

Figure 4: Deletion.

2 Dynamic Programming

Algorithmic paradigms.

- Greedy
- Divide-and-conquer
- Dynamic programming

Weighted interval scheduling. Job j starts at s_j, finishes at f_j, and has weight (or value) v_j. Two jobs are compatible if they don't overlap. The goal is to find the maximum weight subset of mutually compatible jobs.

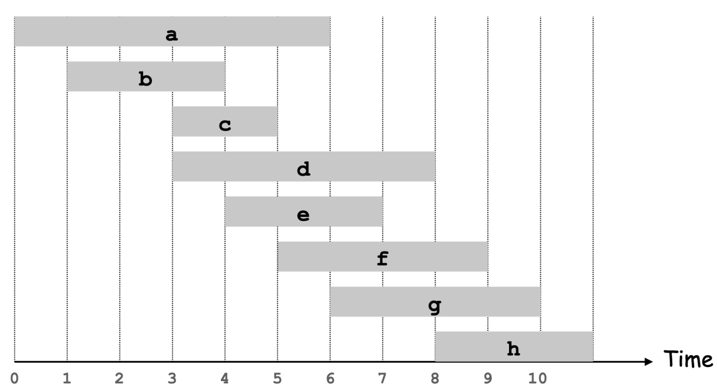

Figure 5: Weighted interval scheduling.

If all weights are 1, the greedy algorithm works, and we pick the job with the earliest finish time that is compatible with chosen jobs.

Without loss of generality, we assume $f_1 \leq \cdots \leq f_n$. Let $p(j)$ be the largets index $i < j$ such that job i is compatbile with j. Let `OPT(j)` be the value of optimal solution to the problem consisting of jobs $1, \ldots, j$.

- **Case I:** OPT selects job j. We collect profit v_j and `OPT(p(j)`.
- **Case II:** OPT does not select job j. We collect profit `OPT(j-1)`.

We have

$$\text{OPT}(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{v_j + \text{OPT}(p(j)), \text{OPT}(j-1)\} & \text{otherwise} \end{cases}$$

Implementation. Store results of sub-problems in a cache to avoid computing sub-problems multiple times. This algorithm takes $O(n \log n)$ time.

- Sort by finish time: $O(n \log n)$
- Computing $p(\cdot)$: $O(n \log n)$ via sorting by start time
- Running time of `OPT` is only $O(n)$