CSCE 629-601 Analysis of Algorithms

Fall 2022

Instructor: Dr. Jianer Chen
Office: PETR 428
Phone: (979) 845-4259
Email: `chen@cse.tamu.edu`
Office Hours: MWF 3:50pm–5:00pm

Teaching Assistant: Vaibhav Bajaj
Office: EABC 107B
Phone: (979) 739-2707
Email: `vaibhavbajaj@tamu.edu`
Office Hours: T; 2pm-3pm, TR: 4pm-5pm

Course Notes #1. 2-3 Trees

A *set* is a collection of *elements*. All elements of a set are different, which means no set can contain two copies of the same element. We will assume that elements of a set are linearly ordered by a relation, usually denoted "<" and read "less than" or "precedes".

Let S be a set and let u be an arbitrary element of a universal set of which S is a subset. The fundamental operations occurring in set manipulation include:

- Search(u, S): Is $u \in S$?
- Insert(u, S): Add the element u to the set S.
- Delete(u, S): Remove the element u from the set S.

When the universal set is linearly ordered, the following operations are also important:

- Min(S): Report the minimum element of the set S.
- Split(u, S): Partition the set S into two sets S_1 and S_2, so that S_1 contains all the elements of S that are smaller than or equal to u, and S_2 contains all the elements of S that are larger than u.
- Splice(S, S_1, S_2): Assuming that all elements in the set S_1 are smaller than any element in the set S_2, form the ordered set $S = S_1 \cup S_2$.

We will introduce a special data structure: 2-3 trees, which represent sets of elements and support the above set operations efficiently.

Definition A *2-3 tree* is a tree such that each non-leaf node has two or three children, and every path from the root to a leaf is of the same length.

The following theorem can be proved using induction on n, and the proof is left to the reader.

Theorem 1 *A 2-3 tree of n leaves has height bounded by* $\log n$.

A linearly ordered set of elements can be stored in a 2-3 tree by placing the elements in the leaves of the tree in such a way that for any non-leaf node w of the tree, all elements stored in (the leaves of) the first child $c1(w)$ of w are less than any elements stored in the second child $c2(w)$ of w, and all elements stored in the second child $c2(w)$ of w are less than any elements stored in the third child $c3(w)$ of w (if w has a third child). The node w also keeps three values for its three children $c1(w)$, $c2(w)$, and $c3(w)$:

- $k1(w)$: the largest element stored in the subtree rooted at $c1(w)$.
- $k2(w)$: the largest element stored in the subtree rooted at $c2(w)$.
- $k3(w)$: the largest element stored in the subtree rooted at $c3(w)$ (if $c3(w)$ exists).

Remark. Strictly speaking, the third value $k3(w)$ is not needed. All algorithms can be implemented without the value $k3(w)$, and without increasing the time complexity. However, we suggest to keep the value $k3(w)$ in implementation, which will make the implementation easier.

1 Searching

The algorithm to search an element in a 2-3 tree is given as follows, where r is the root of the 2-3 tree, and x is the element to be searched in the tree. Note that the algorithm returns "True" when the element x is found in the 2-3 tree, and returns "False" otherwise. Moreover, for a leaf note w, we have used the value $k1(w)$ to record the value of the element stored in the leaf.

```
Algorithm  Search(r, x)
1. If (r is empty) return "False";
2. If (r is a leaf node) return (k1(r) == x);
3. If (k1(r) >= x) return Search(c1(r), x);
   Else If (k2(r) >= x) return Search(c2(r), x);
   Else return Search(c3(r), x).
```

Since the height of a 2-3 tree is $O(\log n)$, and the algorithm simply follows a path in the tree from the root to a leaf, and spends time $O(1)$ on each level, the time complexity of the algorithm `Search` is $O(\log n)$, where n is the number of leaves in the tree.

2 Minimum and Maximum

Suppose that we want to find the minimum element stored in a 2-3 tree rooted at r. Recall that in a 2-3 tree the elements are stored in leaf nodes in *ascending* order from left to right. Therefore the problem is reduced to going down the tree, always selecting the left most link, until a leaf node is reached. This leaf node should contain the minimum element stored in the tree. Evidently, the time complexity of this algorithm is $O(\log n)$ for a 2-3 tree with n leaves.

```
Algorithm  Min(r)
1.  If (r is empty) return "Empty-Tree";
2.  If (r is a leaf) return k1(r);
    Else return Min(c1(r)).
```

Similarly, the maximum element in a 2-3 tree can be found in time $O(\log n)$.

3 Insertion

To insert a new element x into a 2-3 tree T rooted at r, we apply a recursive algorithm that dose two things: (1) insert x into the tree T rooted at r; and (2) report whether this insertion splits the tree T rooted at r into two 2-3 trees.

If the 2-3 tree T has at most one leaf, then the job is easy: (1) if T has no leaf (i.e., T represents an empty set), then we simply make a 2-3 tree that consists of a single node, which is both the root and the leaf of the tree, with a value x. (2) if T has only one leaf of value y, then the tree T is a single-node tree, inserting x into T makes a two-leaf tree, whose values are x and y, respectively, and the leaves are ordered properly.

Now suppose that the 2-3 tree T has a height at least 1 with at least two leaves, then we proceed at first as if we were searching x in the tree T. However, at the level just above the leaves, we start our insertion operation recursively. In general, suppose that we want to add a new child w to a node v in the 2-3 tree T. If v has only two children, we simply make w a new child of v, placing the children in the proper order and updating the information of the node v.

Suppose, however, that v already has three children v_1, v_2, and v_3. Then w would be the fourth child of v. We cannot have a node with four children in a 2-3 tree, so we split the node v into two nodes, which we call v and v'. With the new node v', we can let the first two of $\{v_1, v_2, v_3, w\}$ (in terms of the linear order) be children of v, and let the rest two be children of v'. Now, the node v' is the root of a subtree and should be added as a new child to the parent of v. Thus, the operation now can be recursively done at the level of the parent of v.

One special case occurs when we wind up splitting the root. In that case we create a new root, whose two children are the two nodes into which the old root was split. This is how the number of levels in (i.e., the height of) a 2-3 tree increases.

The above discussion is implemented in the following algorithms, where r is the root of the 2-3 tree to which the element x is to be inserted. Note that when the algorithm `Insert(r, x)` returns, `r` becomes the root of the new 2-3 tree.

```
Algorithm Insert(r, x)
1. If (the tree rooted at r has < 2 leaves)
       process directly; return;
2. AddLeaf(r, x, r’);
3. If (r’ != NULL)
       create a new node v;
       let r and r’ be children of v;
       r = v.
```

The procedure `AddLeaf(r,x,r’)` above is implemented by the following recursive algorithm, which inserts a new element x to the 2-3 tree rooted at r. Moreover, if this insertion causes splitting the node r due to exceeding the number of children, then a new node r' is created to take two of the four children from r. Therefore, if r' is not empty when the procedure returns, then r and r', respectively, are the roots of two 2-3 trees of the same height.

```
Algorithm AddLeaf(r, x, r’)  /* the node r is not a leaf */
1. r’ = NULL;
2. If (r is a parent of leaves)
       If (r has 2 children) add x as a new child of r;
       Else /* r has 3 children */
         order x and the three children of r in the linear order;
         let the first two be children of r, and the rest two be children of r’;
       return;
3. If (k1(r) >= x) v = c1(r);
   Else If (k2(r) >= x or c3(r)==NULL) v = c2(r);
   Else v = c3(r);
4. AddLeaf(v, x, v’);
5. If (v’ == Null) return;
6. If (v’ != NULL and r has 2 children) add v’ as a new child of r;
   Else  /* r has 3 children and v’ is not NULL */
     order v’ and the three children of r in the linear order;
     let the first two be children of r; and the rest two be children of r’;
     return.
```

Analysis: Clearly, the running time of the algorithm `Insert` is dominated by that of the procedure `AddLeaf`, which at each level of the 2-3 tree spends constant time (see steps 1-3, 5-6 of the procedure `AddLeaf`). Since a 2-3 tree with n leaves has a height bounded by $\log n$, we conclude that the algorithm `Insert` runs in time $O(\log n)$.

4 Deletion

Suppose that after deleting a leaf w from a 2-3 tree, the parent v of w is left with only one child. If v is the root, then we delete v and let its sole child be the new root, which gives a valid 2-3 tree. If v is not the root, but the parent p of v has at least four grandchidren, then we can rearrange these grandchidren to make either two or three children for p so that the resulting tree again becomes a valid 2-3 tree.

The worst case is that the parent p of v has only three grandchildren, i.e., the node v has only one sibling v' and v' has only two children. In this case, we transfer the sole child of v to the sibling v', and delete v. This resolves the problem for the nodes at the level of v (v' has three children while v is deleted), but may leave the parent node p with only one child. Should this be the case, we repeat the above process, recursively, with p in place of v.

Summarizing these discussions, we get the algorithm `Delete`, as shown below, where procedure `Delete`() is merely a driver for sub-procedure `Del`() in which the actual work is done.

The variables `done` and `1son` in `Del`() are boolean flags used to indicate successful deletion and to detect the case when a node in the tree has only one child, respectively.

In the worst case we need to traverse a path in the tree from root to a leaf to locate the leaf to be deleted, then from that leaf node to the root (the worst case happens when every non-leaf node on the path has only two children and has only one sibling that has only two children in the original 2-3 tree T). Thus the time complexity of `Delete` algorithm for a 2-3 tree with n leaves is $O(\log n)$.

```
Algorithm Delete(r, x)
1. If (r == Null) return "x not found";
2. If (r is a leaf)
     If (x == k1(r)) r = Null; return "x deleted";
     Else return "x not found";
3. Del(r, x, done, 1son);
4. If (done == false) return "x not found";
5. If (1son == true) r = c1(r); return "x deleted".

Algorithm Del(r, x, done, 1son)
1. done = true; 1son = false;
2. If  (r is a parent of leaves)  process properly and return;
    /* i.e., delete x if it is in the tree; update done and 1son */
3. If (x <= k1(r))  r' = c1(r);
   Else if (x <= k2(r)) or (c3(r) == Null)  r' = c2(r);
   Else  r' = c3(r);
4. Del(r', x, done', 1son');
5. If (done' == false)  done = false; return;
6. If (1son' == true)
     If (r has at least 4 grandchildren)
       reorganize the grandchildren of r so that each of r and its
       children has either 2 or 3 children; return;
     Else
       make r a 1-child node (with 3 grandchildren);
       1son = true; return.
```

5 Splice

Splicing two trees into one big tree is a special case of the more general operation of merging two trees. Splice assumes that all the elements in one of the trees are larger than all those in the other tree. This assumption effectively reduces the problem of merging the trees into “pasting” the shorter tree into a proper position in the taller tree. “Pasting” the shorter tree is actually no more than performing an ”adding a child” operation to a proper node in the taller tree.

To be more specific, let T_1 and T_2 be two 2-3 trees which we wish to splice into a single 2-3 tree T, where all elements in T_1 are smaller than that in T_2. Furthermore, assume that the height of T_1 is less than or equal to that of T_2 so that T_1 is "pasted" to T_2 as a left child of a leftmost node at a proper level in T_2. In the case where the heights are equal, the new tree T can be easily constructed by letting T_1 and T_2 be the two children of the root of T. Otherwise, a node v at a proper level in the tree T_2 is found, and T_1 is inserted as the left child of v. Note that the level of the node v in the tree T_2 is given by (assume the root of T_2 is at level 0):

$$height(T_2) - height(T_1) - 1$$

A more detailed description of the algorithm `Splice` is given as follows.

```
Algorithm Splice(T, T1, T2)
/* Assume all elements in T1 are less than any elements in T2 */
1. h1 = height of T1;  h2 = height of T2;
2. If h1 == h2
    create a root r for T and let T1 and T2 be children of r; return;
3. If h1 < h2  find the leftmost node v in T2 at level (h2 - h1 - 1),
    add T1 as a new child of v;  T = T2;  return;
4. If h1 > h2  find the rightmost node v in T1 at level (h1 - h2 - 1),
    add T2 as a new child of v;  T = T1;  return.
```

Note that steps 3-4 in the algorithm `Splice` may cause other nodes in a 2-3 tree (e.g., the node `v`) to have more than 3 children. Therefore, these steps should really be implemented as recursive procedures that are similar to the algorithm `AddLeaf` as given in the algorithm `Insert`.

The heights `h1` and `h2` of the trees `T1` and `T2`, respectively, in step 1 can be computed by tracing a path in the trees from the root to (any) leaf. Thus, step 1 takes time $O(\log n)$. So the algorithm `Splice` runs in time $O(\log n)$. If we already know the values of `h1` and `h2` so step 1 of the algorithm can be omitted, then the algorithm follows a path in the taller tree from the root to a node a level h, where h is the difference of the heights of the two trees `T1` and `T2` minus 1. Thus, under this assumption, the running time of the algorithm `Splice` will be $O(h)$. This is summarized in the following theorem.

Theorem 2 *The algorithm* `Splice` *takes time* $O(\log n)$. *If the heights of the two trees are known, then the two trees can be spliced in time* $O(h)$, *where* h *is the difference of the heights of the two trees.*

The second part of Theorem 2 will be useful in developing an algorithm for the more complicated operation `Split` on 2-3 trees, which is given in the next section.

6 Split

By splitting a given 2-3 tree T into two 2-3 trees, T_1 and T_2, at a given element x, we mean to split the tree T in such a way that all elements in T that are less than or equal to x go to T_1 while the remaining elements in T go to T_2.

The idea is as follows: based on the way we search the element x in the tree T, we in addition use two stacks to store, respectively, the subtrees to the left and the subtrees to the right of the traversed path (*splitting path*). Finally, the subtrees in each stack are spliced together to form the desired trees T_1 and T_2. The algorithm is given as follows.

```
Algorithm  Split(T, x, T1, T2)
/* Split T into T1 and T2 such that all elements in T1 are <= x, and all
   elements in T2 are > x, where SL and SR are stacks.*/
1. r = the root of T;
2. While r is not a leaf Do
     If (x <= k1(r))
       If (c3(r) != Null) SR <-- c3(r);
       SR <-- c2(r);
       r = c1(r);
     Else If (k1(r) < x <= k2(r))
       SL <-- c1(r);
       If (c3(r) != Null) SR <-- c3(r);
       r = c2(r);
     Else /* x is in the third child of r */
       SL <-- c1(r); SL <-- c2(r);
       r = c3(r);
3. /* r is a leaf */
    If (x <= k1(r)) SL <-- r; Else RL <-- r;
/* construct T1 and T2 */
4. T1 <-- SL;
5. While SL is not empty Do
     t <-- SL;
     Splice(T1, t, T1);
6. T2 <-- SR;
7. While SR is not empty Do
     t <-- SR;
     Splice(T2, T2, t);
```

Note that we have omitted certain special cases in the above algorithm. For example, if x is smaller than all elements in T, then we would have T1 $= \emptyset$ and T2 $=$ T. Similarly we can handle the case where x is larger than all elements in T. These cases can be tested and processed in time $O(\log n)$. We leave the details to the reader.

Suppose that the subtrees in the stack SL are τ_1, τ_2, ..., τ_h, and that the subtrees were pushed into the stack SL in the order τ_h, τ_{h-1}, ..., τ_1. By the properties of a 2-3 tree, we know that for all i, all elements in the subtree τ_i are smaller than any element in the subtree τ_{i-1}. Since the subtrees in SL are popped out from SL in the order of τ_1, τ_2, ..., τ_h and are spliced in the tree T1 (steps 4-5), we know that the splice operation Splice(T1, t, T1) is always valid. Similarly, steps 6-7 are valid.

It is easy to see that the While loop in step 2 takes time $O(\log n)$. The analysis for the rest of the algorithm is a bit more complicated. In each of steps 4-5 and steps 6-7, we may need to splice more than a constant number of subtrees. Thus, if we count the complexity of each splice as $O(\log n)$, we would not be able to bound the running time of these steps by $O(\log n)$.

Note that the heights of the subtrees in the stacks SL and SR can be easily computed while we traverse the splitting path in T from its root in step 2 of the algorithm Split. By taking advantage of this fact and Theorem 2, we can have more precise analysis for the complexity of the algorithm Split.

The use of the stacks SL and SR to store the subtrees guarantees that the height of a subtree closer to stack top is not larger than that of the subtree immediately deeper in the stack. A crucial observation is that since we splice shorter trees first (which are on the top part of the stacks), the difference between the heights of two trees to be spliced is always small. In fact, the total time spent on splicing all these subtrees is bounded by $O(\log n)$. We give a formal proof for this as follows.

Assume before we start step 4, the subtrees stored in the stack SL are

$$\tau_1, \tau_2, \cdots, \tau_h, \tag{1}$$

in the order from top to bottom in the stack SL. For a 2-3 tree τ, denote by $ht(\tau)$ the height of τ. According to the algorithm Split, we have

$$ht(\tau_1) \leq ht(\tau_2) \leq \cdots \leq ht(\tau_h)$$

and no three consecutive subtrees in the stack have the same height. Thus, we can partition the sequence (1) into non-empty "segments" such that each segment contains subtrees of the same height in the sequence:

$$s_1, s_2, \cdots, s_q$$

Each s_i either is a single subtree or consists of two consecutive subtrees of the same height in sequence (1). Moreover, $q \leq \log n$. Let $ht(s_i)$ be the height of the subtrees contained in the segment s_i. We have

$$ht(s_1) < ht(s_2) < \cdots < ht(s_q) \tag{2}$$

The While loop in Step 5 first splices the subtrees in the segment s_1 into a single 2-3 tree $T_1^{(1)}$, then recursively splices the subtrees in the segment s_i and the 2-3 tree $T_1^{(i-1)}$ into a 2-3 tree $T_1^{(i)}$, for $i = 2, 3, \ldots, q$. We have the following lemma.

Lemma 3 *For all $2 \leq i \leq q$, $ht(s_{i-1}) \leq ht(T_1^{(i-1)}) \leq ht(s_i)$.*

Proof. The inequality $ht(s_1) \leq ht(T_1^{(1)})$ is obvious since $T_1^{(1)}$ is obtained by splicing subtrees in the segment s_1. For $i > 2$, since $T_1^{(i-1)}$ is obtained by splicing the subtrees in s_{i-1} and the tree $T_1^{(i-2)}$, and the subtrees in s_{i-1} have height $ht(s_{i-1})$, we must have $ht(s_{i-1}) \leq ht(T_1^{(i-1)})$.

Now consider the second inequality. The 2-3 tree $T_1^{(1)}$ is obtained by splicing the subtrees in the segment s_1, which contains at most two subtrees, both of height $ht(s_1)$. Thus, the height of the 2-3 tree $T_1^{(1)}$ is at most $ht(s_1) + 1$, which, by (2), is not larger than $ht(s_2)$. Thus, $ht(T_1^{(1)}) \leq ht(s_2)$, and the second inequality in the lemma holds true for the case $i = 2$.

For the case $i > 2$, the tree $T_1^{(i-1)}$ is obtained by splicing the subtrees in the segment s_{i-1} and the tree $T_1^{(i-2)}$ (note $i > 2$). By the inductive hypothesis, $ht(T_1^{(i-2)}) \leq ht(s_{i-1})$. If the segment s_{i-1} consists of a single subtree τ of height $ht(s_{i-1})$, then splicing the tree τ of height $ht(s_{i-1})$ and the tree $T_1^{(i-2)}$ of height at most $ht(s_{i-1})$ results in a 2-3 tree $T_1^{(i-1)}$ of height at most $ht(s_{i-1}) + 1$, which, by (2), is not larger than $ht(s_i)$.

Now suppose that the segment s_{i-1} consists of two subtrees τ' and τ'' of height $ht(s_{i-1})$, and that $T_1^{(i-2)}$ is first spliced with τ' to result in a tree τ^+, then τ^+ is spliced with τ'' to result in the tree $T_1^{(i-1)}$. The tree τ^+ can have a height either $ht(s_{i-1})$ or $ht(s_{i-1}) + 1$ (note $ht(T_1^{(i-2)}) \leq ht(s_{i-1})$). If the height of τ^+ is $ht(s_{i-1})$, then splicing τ^+ of height $ht(s_{i-1})$ and the tree τ'' (also of height $ht(s_{i-1})$) results in the tree $T_1^{(i-1)}$ of height at most $ht(s_{i-1}) + 1 \leq ht(s_i)$. If the height of the tree τ^+ is $ht(s_{i-1})+1$, then the root of the tree τ^+ must have only two children (see algorithm Insert, step 3). Thus, splicing τ^+ and τ'' will not increase the tree height (see algorithm AddLeaf, step 6), so the tree $T_1^{(i-1)}$ resulted from the splicing has height $ht(s_{i-1}) + 1$, again not larger than $ht(s_i)$. This concludes that we will always have $ht(T_1^{(i-1)}) \leq ht(s_i)$. The lemma is proved. □

Now we are ready for the following theorem

Theorem 4 *The algorithm* `Split` *runs in time* $O(\log n)$.

PROOF. It is obvious that steps 1, 2, 3, 4, and 6 of the algorithm `Split` take time $O(\log n)$. Thus, to prove the theorem, we only need to prove that the `While` loops in steps 5 and 7 of the algorithm take time $O(\log n)$.

We first consider, for each i, the amount of time spent on splicing the 2-3 tree $T_1^{(i-1)}$ and the subtrees in the segment s_i to get the 2-3 tree $T_1^{(i)}$. By Lemma 3, $ht(T_1^{(i-1)}) \leq ht(s_i)$. If s_i is a single subtree τ_i, then by Theorem 2, the time for splicing $T_1^{(i-1)}$ and τ_i to get $T_1^{(i)}$ is bounded by a constant times $ht(s_i) - ht(T_1^{(i-1)})$.

Now suppose that s_i consists of two subtrees τ_i' and τ_i'', and that the tree $T_1^{(i-1)}$ is first spliced with τ_i' that gives a tree τ_i^+, then the tree τ_i^+ is spliced with τ_i'' to get $T_1^{(i)}$. The time for splicing $T_1^{(i-1)}$ and τ_i' to get τ_i^+ is again bounded by a constant times $ht(s_i) - ht(T_1^{(i-1)})$. Moreover, the height of the resulting tree τ_i^+ is either $h(s_i)$ or $h(s_i) + 1$. So splicing τ_i^+ with τ_i'' of height $ht(s_i)$ takes only constant time. Therefore, in this case, the total time to construct $T_1^{(i)}$ from $T_1^{(i-1)}$ and s_i is bounded by a constant times $ht(s_i) - ht(T_1^{(i-1)}) + 1$.

In summary, to construct the 2-3 tree $T_1 = T_1^{(q)}$, the time of the `While` loop in step 5 of the algorithm `Split` (noticing that the tree $T_1^{(1)}$ can always be constructed from s_1 in constant time) is bounded by a constant times

$$\sum_{i=2}^{q} (ht(s_i) - ht(T_1^{(i-1)}) + 1)$$

By Lemma 3, $ht(s_{i-1}) \leq ht(T_1^{(i-1)})$ for all i. Thus, the time complexity of the `While` loop in step 5 is bounded by a constant times

$$\sum_{i=2}^{q} (ht(s_i) - ht(s_{i-1}) + 1) = ht(s_q) - ht(s_1) + (q-1)$$

Since the quantities $h(s_q)$, $h(s_1)$, and q are all bounded by $\log n$, we conclude that the `While` loop in step 5 takes time $O(\log n)$. The same conclusion applies to step 7 of the algorithm, thus completing the proof of the theorem. □