CST207
DESIGN AND ANALYSIS OF ALGORITHMS

Lecture 10: The Searching Problem

Lecturer: Dr. Yang Lu

Email: luyang@xmu.edu.my

Office: A1-432

Office hour: 2pm-4pm Mon & Thur

Outlines

- Lower Bounds for the Searching Problem
- Interpolation Search
- Searching in Trees
- The Selection Problem

The Searching Problem

- Like sorting, searching is one of the most useful applications in computer science.
- The problem is usually to retrieve an entire record based on the value of some key field.
  - Recall the phonebook example in Lecture 1.
- In this lecture:
  - We analyze the searching problem and show that we have obtained searching algorithms whose time complexities are about as good as our lower bounds.
  - We discuss the data structures used by the algorithms and to discuss when a data structure satisfies the needs of a particular application.

LOWER BOUNDS FOR THE SEARCHING PROBLEM

Lower Bounds for the Search Problem

- The searching problem is that given an array S containing n keys and a key x:
  - Find an index i such that $x = S[i]$ if x equals one of the keys.
  - If x does not equal one of the keys, report failure.
- Binary Search is very efficient for solving this problem when the array is sorted.
  - Recall that its worst-case time complexity is $\lfloor \lg n \rfloor + 1$.
  - Can we improve on this performance?
- As long as we limit ourselves to algorithms that **search only by comparisons of keys**, such an improvement is not possible.

Lower Bounds for the Search Problem

Theorem 1

Any deterministic algorithm that searches for a key x in an array of n distinct keys only by comparisons of keys must in the worst case do at least

$$\lfloor \lg n \rfloor + 1 \text{ comparisons of keys.}$$

Theorem 2

Among deterministic algorithms that search for a key x in an array of n distinct keys only by comparison of keys, Binary Search is optimal in its average-case performance if we assume that x is in the array and that all array slots are equally probable. Therefore, under these assumptions, any such algorithm must on the average do at least approximately

$$\lfloor \lg n \rfloor - 1 \text{ comparisons of keys.}$$

Lower Bounds for the Search Problem

- We established that Binary Search is optimal in its average-case performance **given specific assumptions about the probability distribution**.
- For other probability distributions, it may not be optimal.
  - For example, if it is known that the probability was 0.9999 that x equaled $S[10]$, it would be optimal to compare x with $S[10]$ first.
  - Recall the situation and solution for optimal binary search tree.

INTERPOLATION SEARCH

Use Extra Information for Searching

- The bounds just obtained are for algorithms that rely only on comparisons of keys.
  - We can improve on these bounds if we use some other information to assist in our search.
- Recall that you can more than just compare keys to find Lisa Barber's number in the phone book.
  - You don't start in the middle of the phone book, because you know that the B's are near the front.
  - Your "interpolate" and starts near the front.

Interpolation Search

- Suppose we are searching 10 integers, and we know that:
  - The first integer ranges from 0 to 9.
  - The second integer ranges from 10 to 19.
  - The third integer ranges from 20 to 29.
  - …
  - The tenth from 90 to 99.
- Then we can immediately report failure if the search key x is less than 0 or greater than 99.
- If neither of these is the case, we need only compare x with $S[1 + \lfloor x/10 \rfloor]$.
  - For example, we would compare $x = 25$ with $S[1 + \lfloor 25/10 \rfloor] = S[3]$.
  - If they were not equal, we would report failure.

Interpolation Search

- We usually do not have this much information.
- However, in some applications it is reasonable to assume that the keys are **close to being evenly distributed** between the first one and the last one.
  - If you know it, you can use it to accelerate searching.
- In such cases, instead of checking whether x equals the middle key, we can check whether x equals the key that is located about where we would expect to find x.
  - For example, if we think 10 keys are close to being evenly distributed from 0 to 99, we would expect to find $x = 25$ about in the third position, and we would compare x first with $S[3]$ instead of $S[5]$.

Interpolation Search

- The algorithm that implements this strategy is called *Interpolation Search*.
- We use linear interpolation to determine approximately where we feel x should be located:

$$mid = \underbrace{low}_{\text{start index}} + \left\lfloor \underbrace{\frac{x - S[low]}{S[high] - S[low]}}_{\text{position in [0,1]}} \times \underbrace{(high - low)}_{\text{index range}} \right\rfloor.$$

  - For example, if $S[1] = 4$ and $S[10] = 97$, and we were searching for $x = 25$, $mid = 1 + \left\lfloor \frac{25-4}{97-4} \times (10 - 1) \right\rfloor = 3$.

Pseudocode for Interpolation Search

- It can be shown that the average-case time complexity of Interpolation Search is given by

$$A(n) \approx \lg(\lg n)\,.$$

- If n equals one billion (10^{10}), $\lg(\lg n)$ is about 5, whereas $\lg n$ is about 30.

```
void interpolation_search (int n,
                           const number S[],
                           number x,
                           index & i)
{
    index low, high, mid;
    number denominator;

    low = 1; high = n; i = 0;
    if (S[low] <= x <= S[high])
        while (low <= high && i == 0){
            denominator = S[high] − S[low];
            if (denominator == 0)
                mid = low;
            else
                mid = low + ⌊((x − S[low]) ∗ (high − low)) / denominator⌋;
            if (x == S[mid])
                i = mid;
            else if (x < S[mid])
                high = mid − 1;
            else
                low = mid + 1;
        }
}
```

Drawback

- A drawback of Interpolation Search is its worst-case performance.
- Suppose again that there are 10 keys and their values are 1, 2, 3, 4, 5, 6, 7, 8, 9, and 100.
  - We say that the keys are **close to** being evenly distributed, not **exactly**.
- If $x = 10$, mid would repeatedly be set to low, and x would be compared with every key.
  - At the beginning $mid = 1 + \left\lfloor \frac{10-1}{100-1} \times (10-1) \right\rfloor = 1$. Then it is increased one by one.
- In the worst case, Interpolation Search degenerates to a sequential search.
  - The average-case time complexity of Interpolation Search is good, how can we improve the worst-case?

Robust Interpolation Search

- A variation of Interpolation Search called *Robust Interpolation Search* remedies this situation by establishing a variable gap such that $mid - low$ and $high - mid$ are always greater than gap.
  - The idea is simply to make mid not too close to low or $high$, which leads to sequential search.
- Initially we set

$$gap = \lfloor (high - low + 1)^{1/2} \rfloor.$$

- And we compute mid using the previous formula for linear interpolation. After that computation, the value of mid is updated with the following computation:

$$mid = \min(high - gap, \max(mid, low + gap)) .$$

- For the previous example, $gap = 3$, mid is initialized as 1, and updated to $\min(10 - 3, \max(1{,}1 + 3)) = 4$.

Robust Interpolation Search

- Under the assumptions that the keys are uniformly distributed and that the search key x is equally likely to be in each of the array slots, the average-case time complexity for Robust Interpolation Search is in $\Theta(\lg(\lg n))$.
- Its worst-case time complexity is in $\Theta\left((\lg n)^2\right)$, which is worse than Binary Search but much better than Interpolation Search.
  - We keep the same average-case time complexity but improve the worst-case time complexity.

SEARCHING IN TREES

Search Trees

- Even though Binary Search and its variations, Interpolation Search and Robust Interpolation Search, are very efficient, they cannot be used in many applications because **an array is not an appropriate structure for storing the data in these applications**.
- Then we show that a tree is appropriate for these applications.
- Furthermore, we show that we have $\Theta(\lg n)$ algorithms for searching trees.

Static and Dynamic Searching

- By *static searching* we mean a process in which the records are all added to the file at one time and there is **no need to add or delete records later**.
  - An example of a situation is the searching done by operating systems commands.
- Many applications, however, require *dynamic searching*, which means that **records are frequently added and deleted**.
  - An airline reservation system is an application that requires dynamic searching, because customers frequently call to schedule and cancel reservations.

Static and Dynamic Searching

- An array structure is inappropriate for dynamic searching, because when we add a record in sequence to a sorted array, we must move all the records following the added record.
- Although we can readily add and delete records using a linked list, there is no efficient way to search a linked list.
  - You can't use index to access nodes in a linked list.
- Dynamic searching can be implemented efficiently using a tree structure.
  - First we discuss binary search trees. After that, we discuss B-trees, which are an improvement on binary search trees. B-trees are guaranteed to remain balanced.

Binary Search Tree

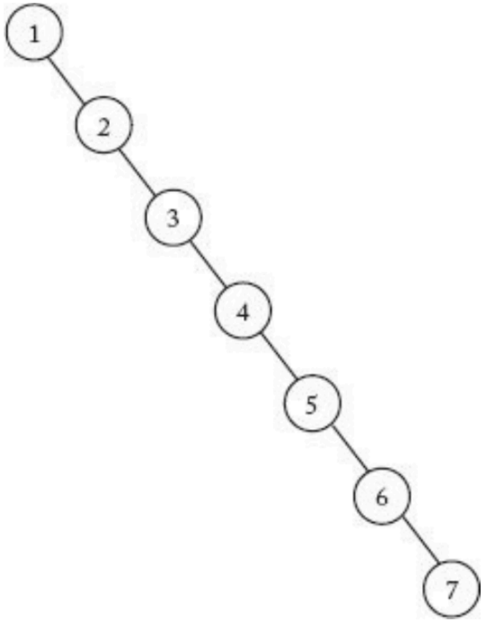
- We have used dynamic programming to solve the problem of optimal binary search tree when keys have different probabilities.
- However, the purpose there was to discuss a static searching application.
  - The algorithm that builds the tree requires that all the keys be added at one time.
  - After creation of the optimal binary search tree, we will use it for a while without changing it.

Binary Search Tree and Binary Search

- Recall that the keys are ordered by an *in-order traversal* of a binary tree.
  - Traverse the tree by first visiting all the nodes in the left subtree, then visiting the root, and finally visiting all the nodes in the right subtree.
- When we search a key in a search tree, we actually do the same sequence of comparisons as done by Binary Search.
  - Therefore, like Binary Search, searching in a search tree is optimal for searching n keys when the tree is balanced.

Skewed Tree

- The drawback of binary search trees is that when keys are dynamically added and deleted, there is no guarantee that the resulting tree will be balanced.
  - For example, if the keys are all added in increasing sequence, we obtain the tree in the figure.
- This tree, which is called a *skewed tree*, is simply a linked list.
  - Search any key to this tree results in a sequential search.
  - In this case, we have gained nothing by using a binary search tree instead of a linked list.



A skewed tree

Image source: Figure 8.6, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Average Search Time

- If the keys are added at random, intuitively it seems that the resulting tree will be closer to a balanced tree much more often than it will be closer to a linked list.
  - Bad luck happens rarely. For most of the time, we face to the case that is neither good luck (balanced tree) nor bad luck (skewed tree).

Theorem 3

Under the assumptions that all inputs are equally probable and that the search key x is equally probable to be any of the n keys, the average search time over all inputs containing n distinct keys, using binary search trees, is given approximately by

$$A(n) \approx 1.38 \lg n \,.$$

Balance Search Tree

- To solve the problem of skewed tree, one solution is to write a balancing program that takes as input an existing binary search tree and outputs a balanced binary search tree containing the same keys.
- The program is then run periodically.
- The optimal binary search tree by dynamic programming is an algorithm for such a program.
  - That algorithm is more powerful than a simple balancing algorithm because it is able to consider the probability of each key being the search key.

Balance Search Tree

- The the data amount is huge, periodically rebuilding the whole tree is time-consuming.
- In a very dynamic environment, it would be better if the tree never became skewed in the first place.
- Such balanced binary tree is called *AVL tree*, in which a binary tree is maintained balanced when adding and deleting nodes.
  - The insertion and deletion times for these algorithms are guaranteed to be $\Theta(\lg n)$, as is the search time.
  - Transfer the computation to insertion and deletion from searching in the worst-case.

B-Tree

- Another solution to maintain balanced binary search tree is *B-tree*.
- A B-tree T is a rooted tree (whose root is $root[T]$) with the following properties:
  - Every node s has the following fields:
    - $n[s]$: the number of keys currently stored in node s.
    - The $n[s]$ keys themselves, stored in nondecreasing order, so that

$$key_1[s] \leq key_2[s] \leq \ \ldots \leq key_{n[s]}[s]$$

  - Each internal node s also contains $n[s]+1$ pointers $c_1[s], c_2[s], \ldots, c_{n[s]+1}[s]$ to its children.

B-Tree

- The keys $key_i[s]$ separate the ranges of keys stored in each subtree: if x_i is any key stored in the subtree with root $c_i[s]$, then

$$x_1 \le key_1[s] \le x_2 \le key_2[s] \le \ \dots \le key_{n[s]}[s] \le k_{n[s]+1}$$

- In this figure:
  - For the root: $key_1[root] = 10, key_2[root] = 15.$
  - For each subtree with root:
    - $key_1\left[c_1[root]\right] = 5.$
    - $key_1\left[c_2[root]\right] = 13.$
    - $key_1\left[c_3[root]\right] = 17,\ key_2\left[c_3[root]\right] = 20,\ key_3\left[c_3[root]\right] = 30.$

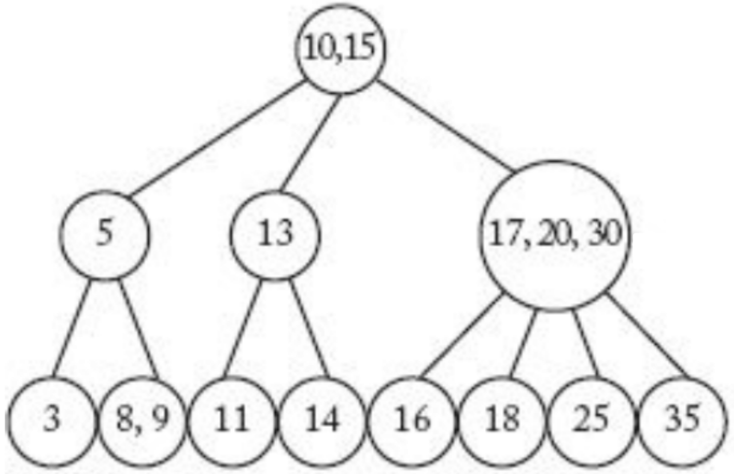


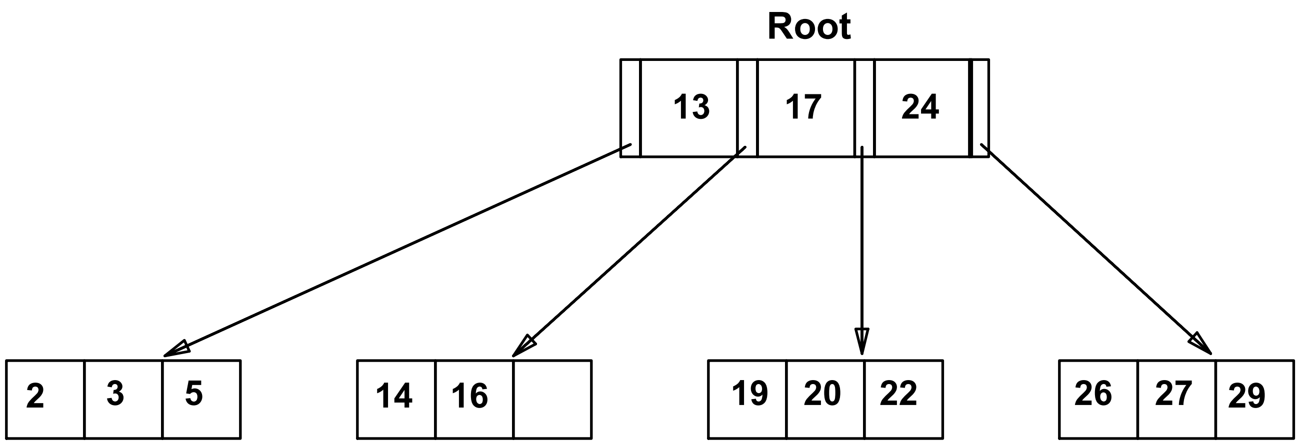
Image source: Figure 8.7, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

B-Tree

- In a B-tree, all leaves have the same depth, which is the tree's height h.
- There are lower and upper bounds on the number of keys a node can contain:
  - t: minimum degree of the B-tree.
  - Every node other than the root must have at least $t - 1$ keys (i.e., a non-root internal node must have at least t children).
  - Every node can contain at most $2t - 1$ keys (i.e., an internal node can have at most $2t$ children).
  - We say that a node is *full* if it contains exactly $2t - 1$ keys.

An Example of B-Tree

- Search begins at root, and key comparisons direct it to a leaf.
- For example, if the search key is 20, it will compare 13, 17, 24, 19, 20.



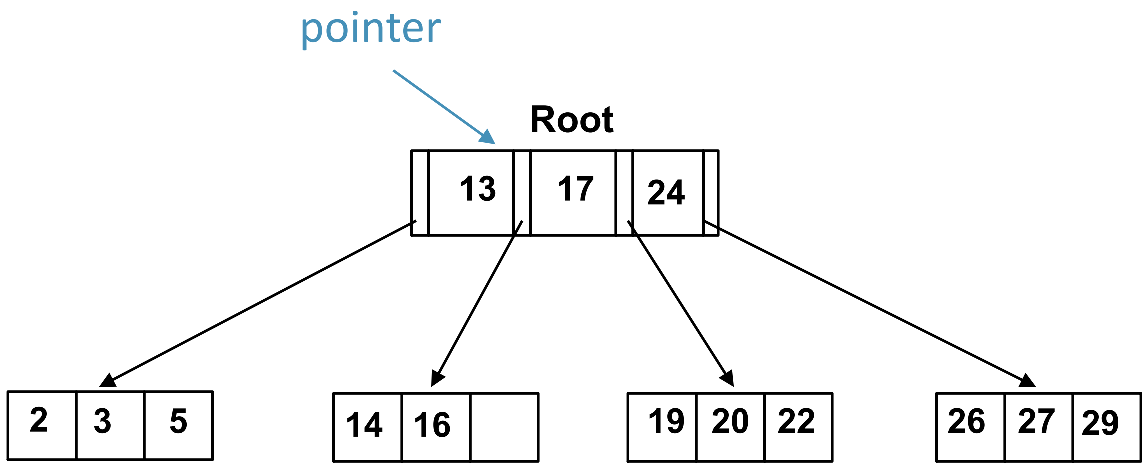
A B-tree with degree $t = 2$

Key Inserting for B-Tree

- **Step 1**: If the root is full, split the root and a node containing the middle key becomes the new root.
- **Step 2**: Direct to the subtree where the search key should be. Suppose the node s is the root of the subtree we are pointing to.
  - If s is full, must split s. Redistribute entries evenly, and move up middle key.
  - If s is a leaf, then insert a key into s. Done! Otherwise, go to Step 2.

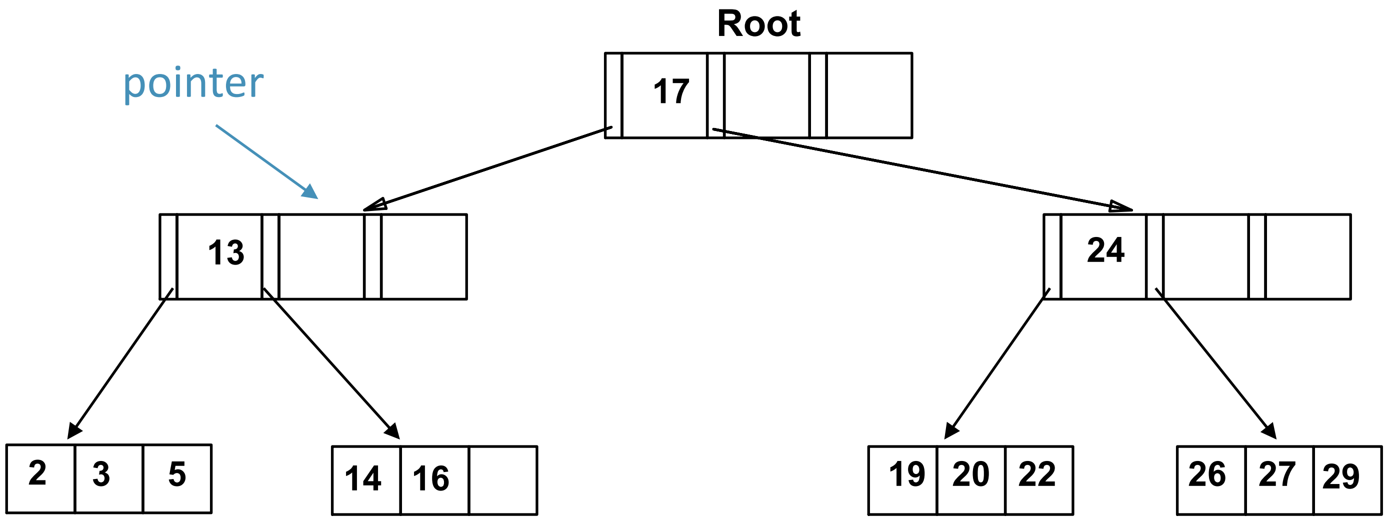
Example of Key Inserting for B-Tree

- We are going to insert the key 8.
- We check the root with Step 1.
  - The root is full, split it.



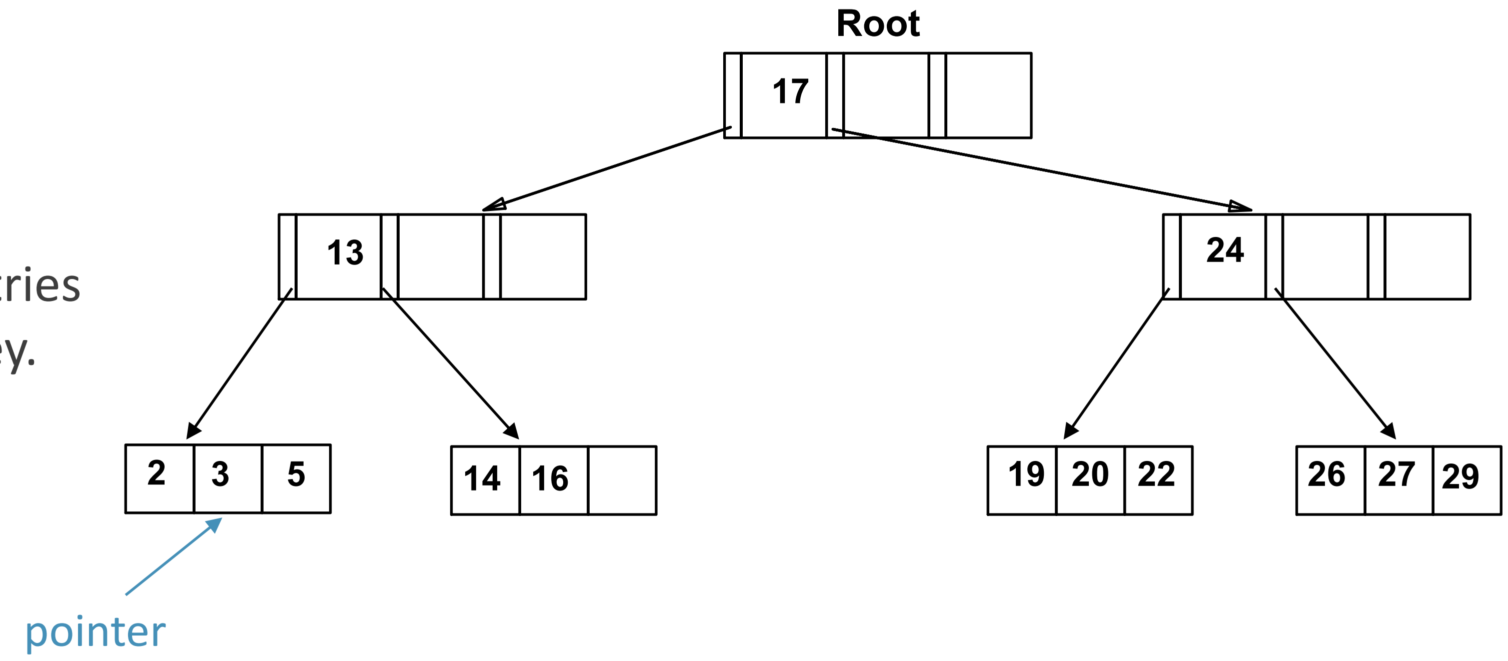
Example of Key Inserting for B-Tree

- After split the root, we go to Step 2.
- Now, the node is neither full, nor a leaf.
- We go to Step 2 again.



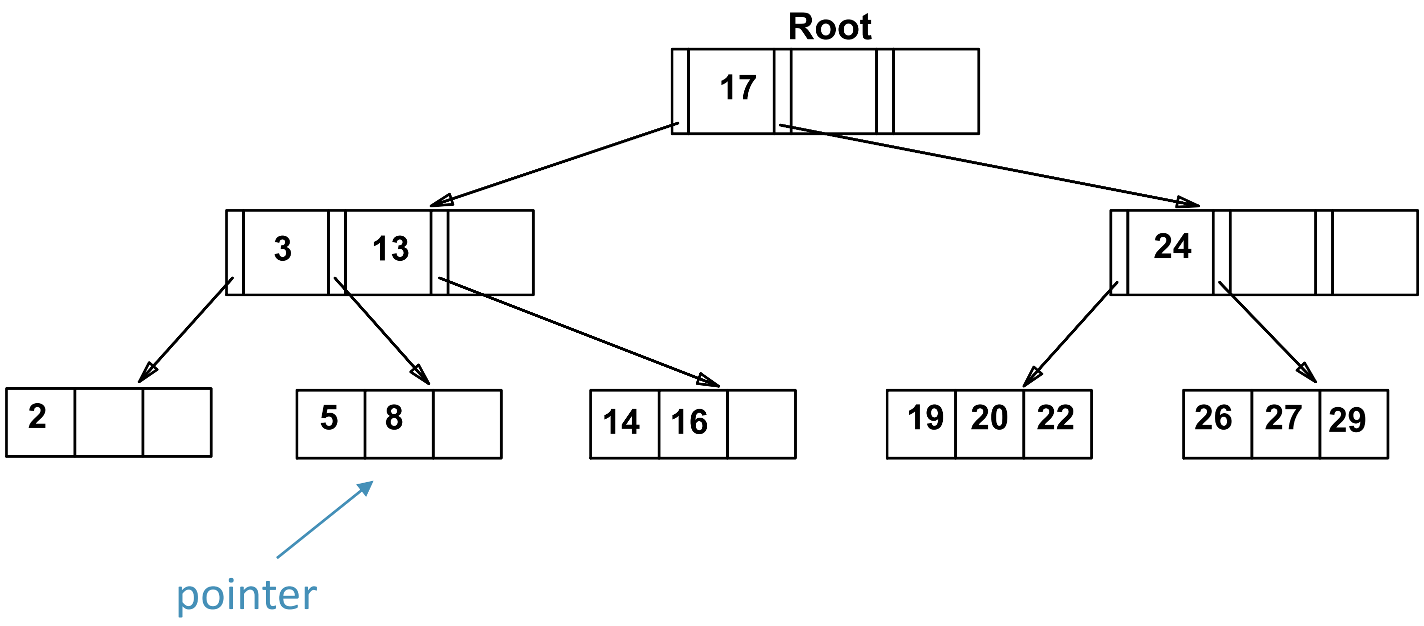
Example of Key Inserting for B-Tree

- Now, the node is full.
  - Split the node, redistribute entries evenly, and move up middle key.



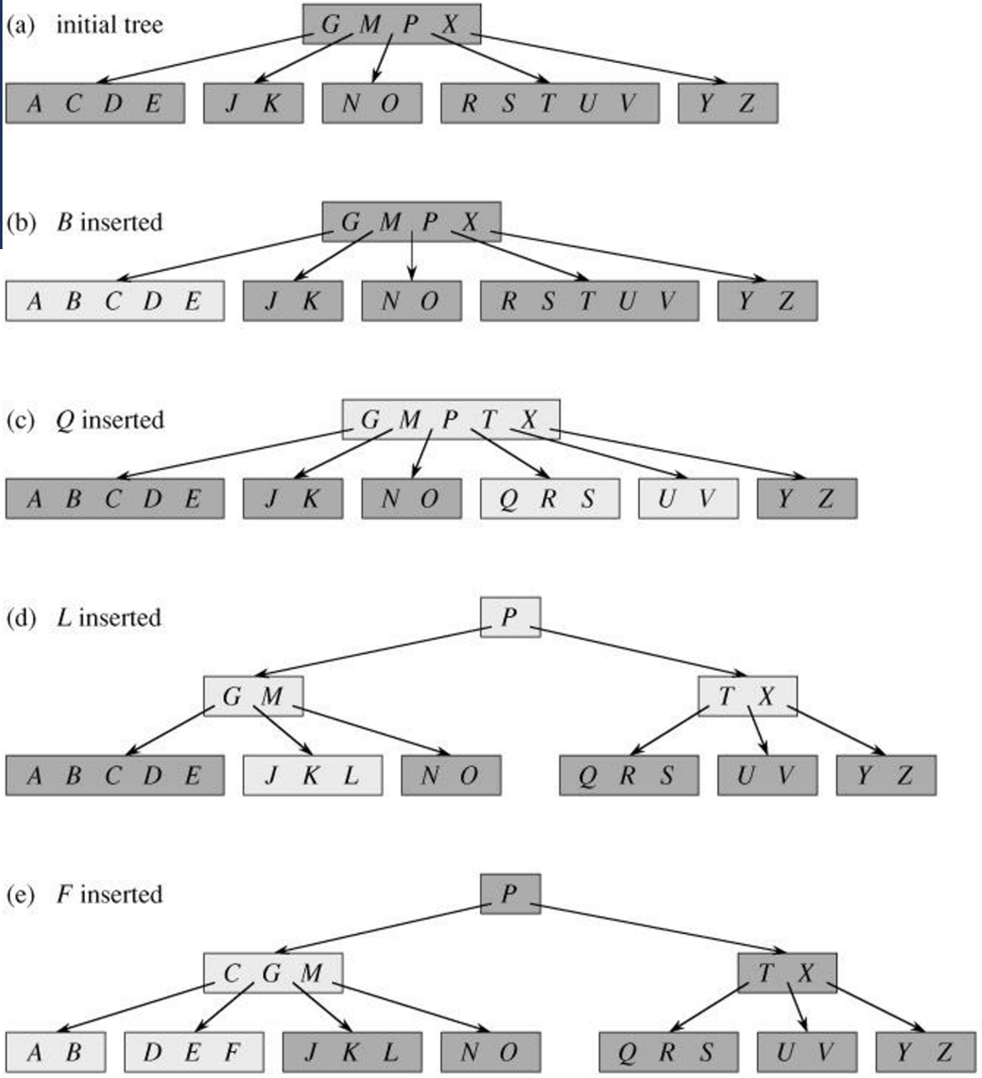
Example of Key Inserting for B-Tree

- Now, the node is a leaf.
  - Insert 8. Done!



Example of Key Inserting for B-Tree

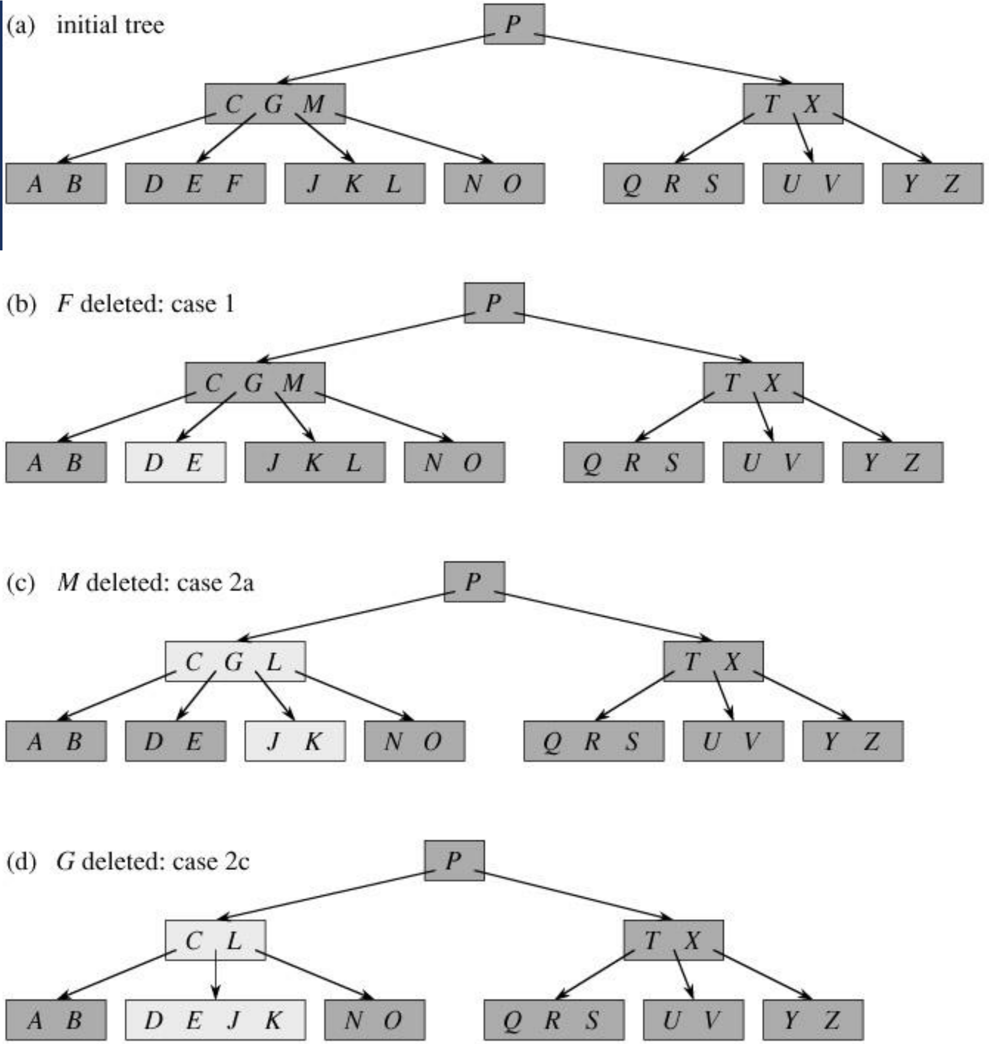
- Summary of key inserting for B-tree: whenever you visit a node, check if it is full first, if yes, split.
- Time complexity for insertion: $O(\lg n)$.



Another example with degree $t = 3$

Key Deletion for B-Tree

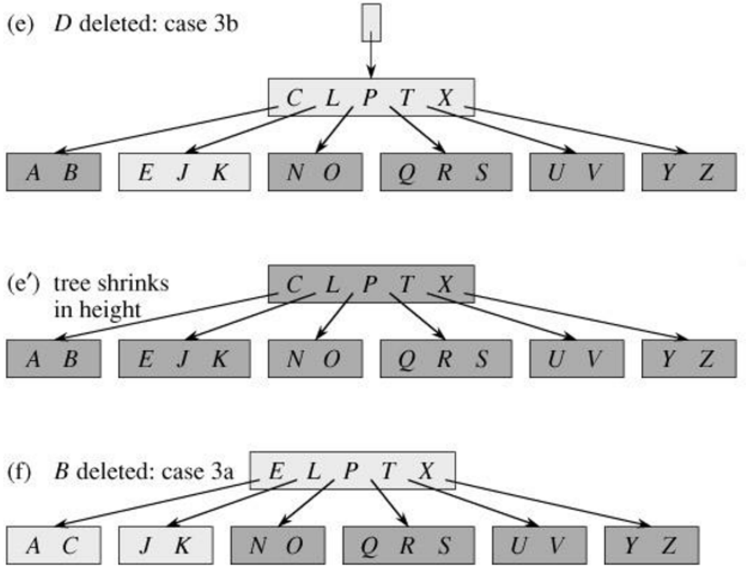
- **Case 1**: If the key x is in node s that has at least t keys and s is a leaf, delete the key x from s.
- **Case 2**: If the key x is in node s and s is an internal node, do the following
  - **Case 2a**: If the child p that precedes x in node s has at least t keys, then find the predecessor x' of x in the subtree rooted at p. Recursively delete x', and replace x by x' in s.
  - **Case 2b**: Symmetrically, if the child q that follows x in node s has at least t keys, then find the successor x' of x in the subtree rooted at q. Recursively delete x', and replace x by x' in s.
  - **Case 2c**: Otherwise, if both p and q have only $t - 1$ keys, merge x and all of q into p, so that s loses both x and the pointer to q, and p now contains $2t - 1$ keys. Then, free q and recursively delete x from p.



Key deletion example with degree $t = 3$

Key Deletion for B-Tree

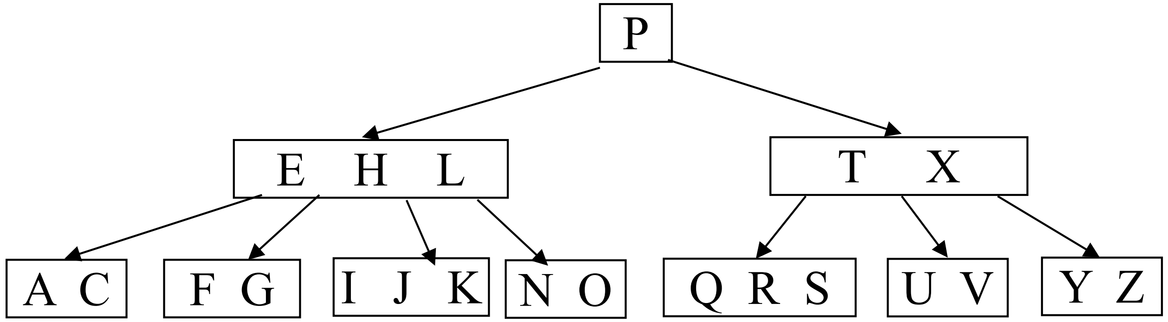
- **Case 3**: If the key x is not present in internal node s, determine the root $c_i[s]$ of the appropriate subtree that must contain x, if x is in the tree at all. If $c_i[s]$ has only $t-1$ keys, execute Case 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then, finish by recursing on the appropriate child of s.
  - **Case 3a**: If $c_i[s]$ has only $t-1$ keys but has an immediate sibling with at least t keys, give $c_i[s]$ an extra key by moving a key from s down into $c_i[s]$, moving a key from $c_i[s]$'s immediate left or right sibling up into s, and moving the appropriate child pointer from the sibling into $c_i[s]$.
  - **Case 3b**: If $c_i[s]$ and both of $c_i[s]$'s immediate siblings have $t-1$ keys, merge $c_i[s]$ with one sibling, which involves moving a key from s down into the new merged node to become the median key for that node.



Key deletion example with degree $t = 3$

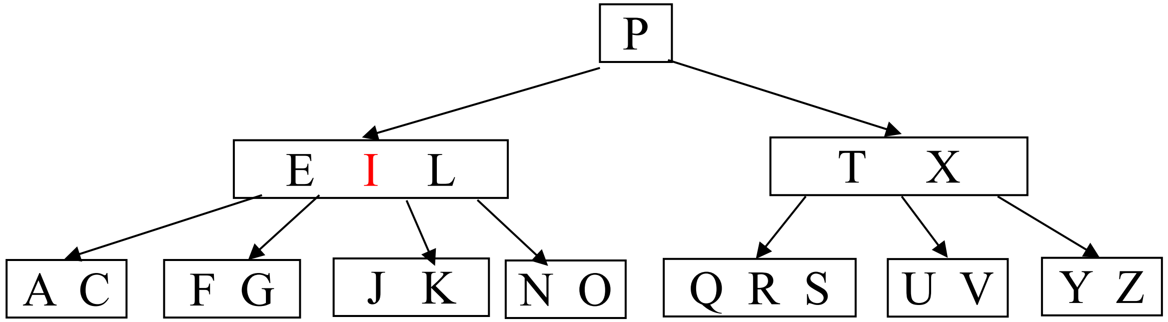
Example of Key Deletion for B-Tree

- Show the results of deleting H, L, T and G, in order, from the B-tree below with degree $t = 3$.



- Delete H (case 2b):

Case 2b: Symmetrically, if the child q that follows x in node s has at least t keys, then find the successor x' of x in the subtree rooted at q. Recursively delete x', and replace x by x' in s.



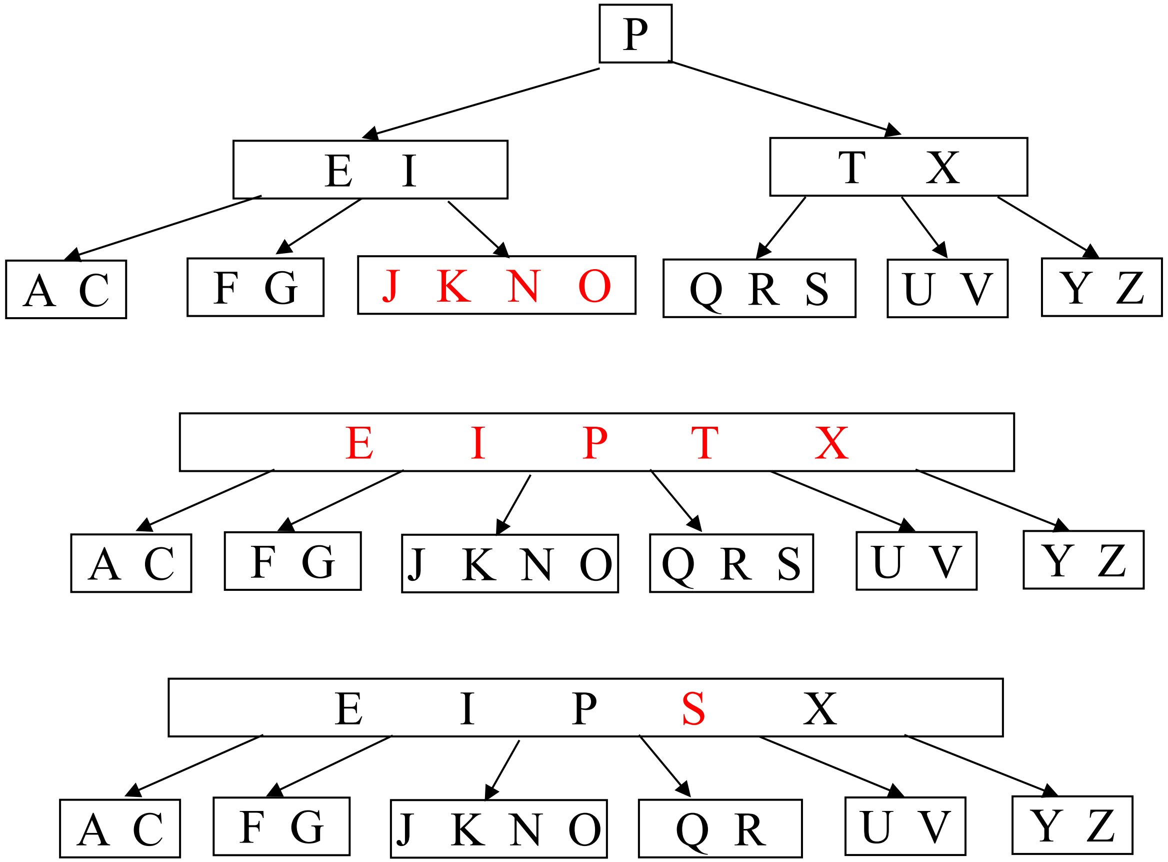
Example of Key Deletion for B-Tree

- Delete L (case 2c):

  Case 2c: Otherwise, if both p and q have only $t-1$ keys, merge x and all of q into p, so that s loses both x and the pointer to q, and p now contains $2t-1$ keys. Then, free q and recursively delete x from p.

- Delete T (case 3b then case 2a):

  Case 3b: If $c_i[s]$ and both of $c_i[s]$'s immediate siblings have $t-1$ keys, merge $c_i[s]$ with one sibling, which involves moving a key from s down into the new merged node to become the median key for that node.

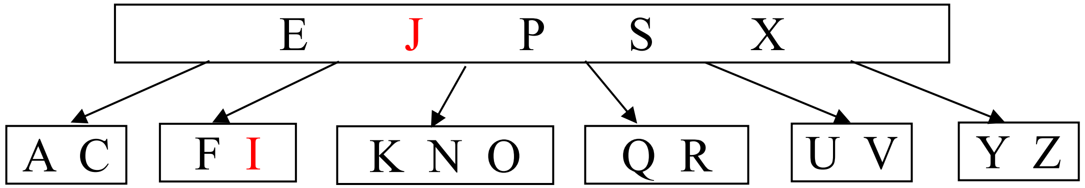


Case 2a: If the child p that precedes x in node s has at least t keys, then find the predecessor x' of x in the subtree rooted at p. Recursively delete x', and replace x by x' in s.

Example of Key Deletion for B-Tree

- Delete G (case 3b):

Case 3b: If $c_i[s]$ and both of $c_i[s]$'s immediate siblings have $t-1$ keys, merge $c_i[s]$ with one sibling, which involves moving a key from s down into the new merged node to become the median key for that node.



Key Deletion for B-Tree

- The actual checking order:
  - Case 3 (before reaching the node where the key is): make sure the root of the subtree containing at least t keys.
  - Case 2 (reaching the internal node where the key is): move one key up or merge.
  - Case 1 (reaching the leaf where the key is): directly delete.
- Time complexity for deletion: $O(\lg n)$.

THE SELECTION PROBLEM

The Selection Problem

- So far we've discussed searching for a key x in a list of n keys.
- Next we address a different searching problem called the *selection problem*.
- This problem is to find the kth-largest (or kth-smallest) key in a list of n keys.
- We assume that the keys are in an unsorted array (the problem is trivial for a sorted array).

The Selection Problem

- One way to find the kth-smallest key in $\Theta(n \lg n)$ time is to sort the keys and return the kth key.
- We develop a method call *Quickselect* that requires fewer comparisons.
- Recall that procedure `partition` in Quicksort partitions an array so that all keys smaller than some pivot item come before it in the array and all keys larger than that pivot item come after it.
- The slot at which the pivot item is located is called the *pivotpoint*.
- We can solve the Selection problem by partitioning until the pivot item is at the kth slot.

Quickselect

- We do this by recursively partitioning :
  - the left subarray if k is less than `pivotpoint`,
  - the right subarray if k is greater than `pivotpoint`.
- When k is equal to `pivotpoint`, we're done.

```
keytype selection (index low, index high, index k)
{
    index pivotpoint;

    if (low == high)
        return S[low];
    else{
        partition(low, high, pivotpoint);
        if (k == pivotpoint)
            return S[pivotpoint];
        else if (k < pivotpoint)
            return selection(low, pivotpoint - 1, k);
        else
            return selection(pivotpoint + 1, high, k);
    }
}
```

```
void partition (index low,
                index high,
                index& pivotpoint)
{
    index i, j;
    keytype pivotitem;

    pivotitem = S[low];
    j = low;

    for (i = low + 1; i <= high; i++)
        if (S[i] < pivotitem){
            j++;
            exchange S[i] and S[j];
        }
    pivotpoint = j
    exchange S[low] and S[pivotpoint];
}
```

Worst-Case Time Complexity

- As in Quicksort, the worst case occurs when the input to the recursive call is $n - 1$.
- This happens, for example, when the array is sorted in increasing order and $k = n$.
- Quickselect therefore has the same worst-case time complexity as Quicksort:

$$W(n) = \frac{n(n-1)}{2}$$

Average-Case Time Complexity

- It can be shown that the average-case time complexity of Quickselect is:

$$A(n) \approx 3n.$$

- On the average, Quickselect does only a linear number of comparisons.
  - The reason is that Quicksort has two calls to `partition`, whereas Quickselect has only one.

Improve Worst-Case Performance

- Similar to Quicksort, the worst-case time complexity of Quickselect can be improved by:
  - Median-of-3 pivot.
  - Random pivot.

Conclusion

After this lecture, you should know:

- What is the lower bound for the search problem if we only compare keys.
- What is the condition to use and how to use interpolation search.
- Why do we need B-tree.
- How does insertion and deletion of B-tree work.
- How to solve the selection problem in linear time.

Assignment

- No tutorial this week. Just implementing Quickselect in Python and submit to Attendance Quiz. You may try to evaluate whether Quickselect has linear time complexity.
- Assignment 5 is released. The deadline is **18:00, 29th June**.

Thank you!

- Any question?
- Don’t hesitate to send email to me for asking questions and discussion. ☺