# Data Structures

Gabriel Istrate

May 13, 2024

Why did we want dynamic sets to start with ?

- To improve algorithms.
- First part of today: complete Computational geometry

Second part: Augmenting Data Structures (e.g. Red-Black Trees)

- What happens if a data structure doesn't support all the operations you want ?
- Augment it: modify it to support the new operations.
- Might need to add additional fields. These need to be maintained.

- Convex hull of a set of points: smallest convex polygon that contains the set of points.
- place elastic rubber band around set of points and let it shrink.
- Two algorithms: Graham's Scan $O(n \log n)$.
- Jarvis's March $O(n \cdot h)$, h the number of points on the convex hull.
- Other algorithms:
- Incremental: points sorted from left to right forming sequence $p_1, \ldots, p_n$. At stage i add p_i to convex hull $CH(p_1, \ldots, p_{i-1})$, forming $CH(p_1, \ldots, p_i)$.
- Divide-and-conquer: divide into leftmost n/2 points and rightmost n/2 points. Compute convex hulls and combine them.
- Prune-and-search method.
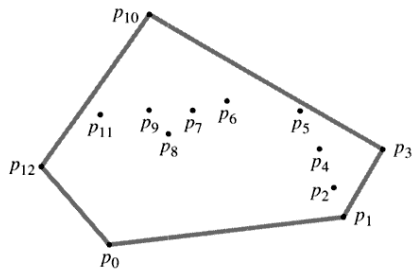
**Figure 33.6** A set of points $Q = \{p_0, p_1, \ldots, p_{12}\}$ with its convex hull CH(Q) in gray.

- Maintains a stack S of candidate points.
- Each point of Q is pushed onto the stack.
- Points not in $CH(Q)$ eventually popped from the stack.
- $TOP(S)$, $NEXT - TO - TOP(S)$: stack functions, do not change its contents.
- Stack returned by the algorithm: points of $CH(Q)$ in counterclockwise order.
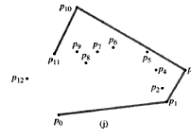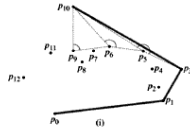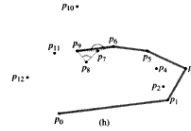
GRAHAM-SCAN(Q)

1 let p_0 be the point in Q with the minimum y-coordinate,
         or the leftmost such point in case of a tie
2 let $\langle p_1, p_2, \ldots, p_m \rangle$ be the remaining points in Q,
         sorted by polar angle in counterclockwise order around p_0
         (if more than one point has the same angle, remove all but
         the one that is farthest from p_0)
3 PUSH(p_0, S)
4 PUSH(p_1, S)
5 PUSH(p_2, S)
6 **for** $i \leftarrow 3$ **to** m
7   **do while** the angle formed by points NEXT-TO-TOP(S), TOP(S),
                 and p_i makes a nonleft turn
8   **do** POP(S)
9   PUSH(p_i, S)
10 **return** S
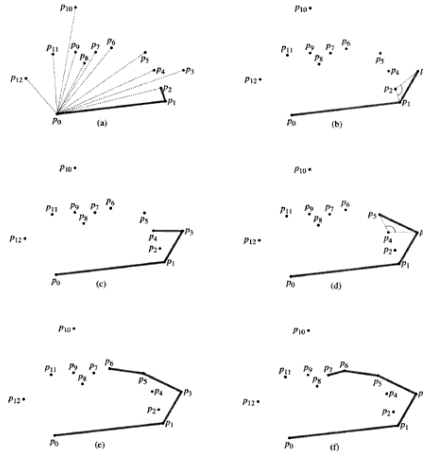
Figure 33.7 The execution of GRAHAM-SCAN on the set Q of Figure 33.6. The current convex hull contained in stack S is shown in gray at each step. (a) The sequence $\langle p_1, p_2, \ldots, p_{12} \rangle$ of points numbered in order of increasing polar angle relative to p_0, and the initial stack S containing p_0, p_1, and p_2. (b)–(k) Stack S after each iteration of the **for** loop of lines 6–9. Dashed lines show nonleft turns, which cause points to be popped from the stack. In part (h), for example, the right turn at angle $\angle p_7 p_8 p_9$ causes p_8 to be popped, and then the right turn at angle $\angle p_6 p_7 p_9$ causes p_7 to be popped. (l) The convex hull returned by the procedure, which matches that of Figure 33.6.
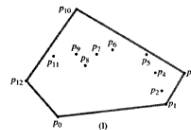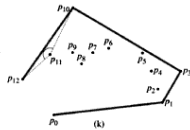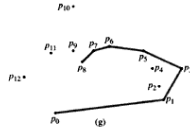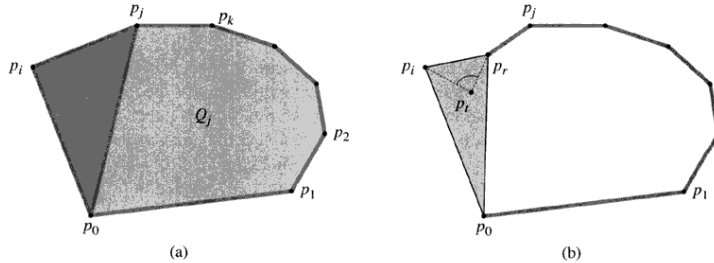
Graham's Scan: Correctness and Performance

- Invariant: at the beginning of each iteration of the for loop stack S contains (from bottom to top) exactly the vertices of $CH(Q_{i-1})$ in counterclockwise order.
- Line 1: $\theta(n)$ time.
- Sorting $\theta(n \log n)$ time.
- Testing for left/right turn: vector product $\theta(1)$ time.
- The rest of the algorithm $O(n)$ time.

Figure 33.8 The proof of correctness of GRAHAM-SCAN. **(a)** Because p_i's polar angle relative to p_0 is greater than p_j's polar angle, and because the angle $\angle p_k p_j p_i$ makes a left turn, adding p_i to $CH(Q_j)$ gives exactly the vertices of $CH(Q_j \cup \{p_i\})$. **(b)** If the angle $\angle p_r p_t p_i$ makes a nonleft turn, then p_t is either in the interior of the triangle formed by p_0, p_r, and p_i or on a side of the triangle, and it cannot be a vertex of $CH(Q_i)$.

- uses a technique known as gift wrapping.
- Simulates wrapping a piece of paper around set Q.
- Start at the same point p_0 as in Graham's scan.
- Pull the paper to the right, then higher until it touches a point. This point is a vertex in the convex hull. Continue this way until we come back to p_0.
- Formally: start at p_0. Choose p_1 as the point with the smallest polar angle from p_0. Choose p_2 as the point with the smallest polar angle from p_1 ...
- ... until we reached the highest point p_k.
- We have constructed the right chain.
- Construct the left chain by starting from p_k and measuring polar angles with respect to the negative x-axis.
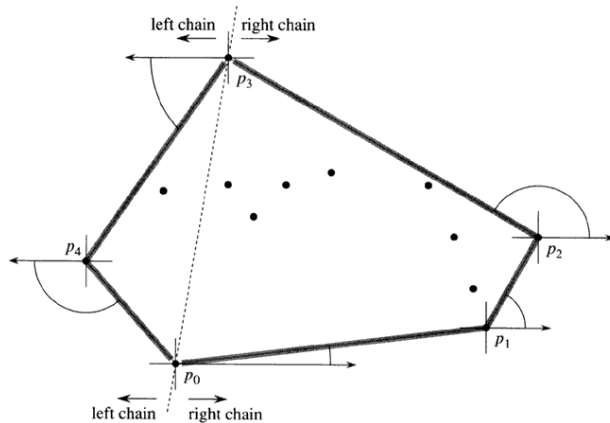
Jarvis's March

Figure 33.9 The operation of Jarvis's march. The first vertex chosen is the lowest point p_0. The next vertex, p_1, has the smallest polar angle of any point with respect to p_0. Then, p_2 has the smallest polar angle with respect to p_1. The right chain goes as high as the highest point p_3. Then, the left chain is constructed by finding smallest polar angles with respect to the negative x-axis.

Augmenting Data Structures

- What if no existing data structure fits your needs ?
- Invent a new one, or ...
- More realistic (in practice): slightly modify a "standard" data structure to support more operations.
- Done by storing extra information in it
- Not always straightforward: new information **must be updated and maintained** by D.S. operations.
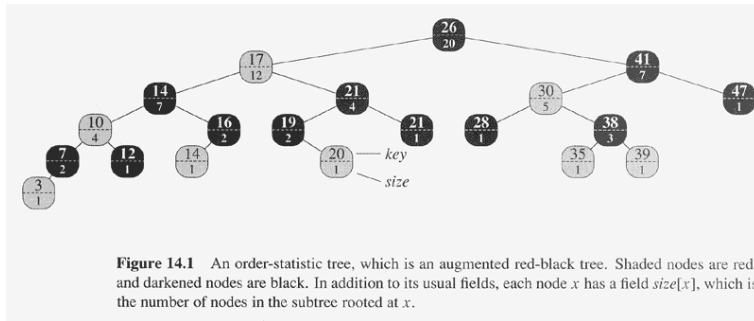
Example: two data structures obtained by modifying red-black trees

- First data structure: supports order statistics queries on a dynamic set.
- Find i'th number in a set or the rank of an element .
- Second data structure: maintain a set of intervals (e.g. time intervals).
- Plus: a general result about augmenting Data Structures.

Dynamic order statistics

- Order statistic tree: red-black tree with one extra field per node: size of the subtree rooted at that node.
- Thus fields: *key*, *color*, *p*, *left*, *right*, *size*.
- $size[nil[T]] = 0$.
- $size[x] = size[left[x]] + size[right[x]] + 1$.
- Supports $OS - SELECT(x, i)$: return i'th smallest element in the tree rooted at x. $O(\log n)$ time.
- Supports $OS - RANK(T, x)$: return the rank of x in the tree T. $O(\log n)$ time.

Order statistics tree

Figure 14.1 An order-statistic tree, which is an augmented red-black tree. Shaded nodes are red, and darkened nodes are black. In addition to its usual fields, each node x has a field $size[x]$, which is the number of nodes in the subtree rooted at x.

Selecting i'th element

- If $i = size[left(x)] + 1$ then (by *BST* property) node x is the i'th element. Return x.
- If $i \leq size[left(x)]$ then node is in $left[x]$. i'th element. Call procedure recursively.
- If $i > size[left(x)] + 1$ then node is in $right[x]$. $i - size[left(x)]$'th element. Call procedure recursively.
- Running time: proportional to the height of the tree: $O(\log n)$.

OS-SELECT(x, i)
1   $r \leftarrow size[left[x]]+1$
2   **if** $i = r$
3   **then return** x
4   **elseif** $i < r$
5   **then return** OS-SELECT($left[x], i$)
6   **else return** OS-SELECT($right[x], i - r$)

```
OS-RANK(T, x)
1   r ← size[left[x]] + 1
2   y ← x
3   while y ≠ root[T]
4       do if y = right[p[y]]
5           then r ← r + size[left[p[y]]] + 1
6           y ← p[y]
7   return r
```

Rank of an element

- Perform inorder traversal.
- Return rank of node x in this traversal.
- Move pointer y from x up towards $root(T)$.
- Maintains the following invariant: at the start of each iteration of the while loop, r is the rank of $key[x]$ in the subtree rooted at y.
- If y is a right child, add the size of its left child to the count.
- Each iteration: $O(1)$ time. y goes up the tree, time complexity $O(\log n)$.

Maintaining subtree sizes: Insertion.

- During LEFT/RIGHT rotations.
- INSERTION. First phase: go from the root to the frontier, inserting the new node as the child of an existing node. new node gets size of 1. Each node from x to the path: size increases by 1. $O(\log n)$.
- Second phase: go up the tree, changing colors, and maintaining the red-black property by rotations.
- Second phase: changes via LEFT/RIGHT rotations.
- LEFT-ROTATE: add lines
- $size[y] \leftarrow size[x]$.
- $size[x] \leftarrow size[left[x]] + size[right[x]] + 1$.
- to rotation pseudocode.
- RIGHT-ROTATE: symmetric.

Maintaining *size* during rotations.

Figure 14.2 Updating subtree sizes during rotations. The link around which the rotation is performed is incident on the two nodes whose *size* fields need to be updated. The updates are local, requiring only the *size* information stored in x, y, and the roots of the subtrees shown as triangles.
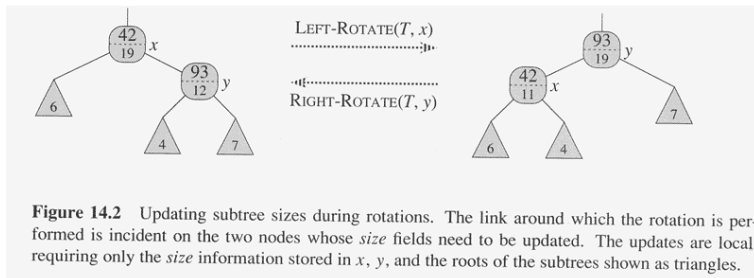
Maintaining subtree sizes: Deletion.

- DELETION: two phases.
- First phase: delete node. Update tree size on the path from the node to the top. Decrement by 1 for each node.
- Rotations: as for insertion.

How to augment a data structure

- Four steps:
- 1. Choose underlying data structure.
- 2. Determine additional information to be maintained.
- 3. Verify that additional information can be maintained in the D.S. operations.
- 4. develop new operations required by new fields.

How to augment a data structure (II)

1. Choose red-black trees. Clue: supports other dynamic set operations on total order: MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR.
2. We didn't need field size to implement OS-SELECT, OS-RANK, but then operations wouldn't run in $O(\log n)$ time. Additional information to be maintained: sometimes pointer rather than data.
3. Ideally only a few elements need to be updated to maintain D.S. E.g. if we simply stored in each node it rank in the tree then OS-SELECT and OS-UPDATE would be efficient but inserting a smallest node causes changes in the whole tree.
4. Developed OS-SELECT, OS-RANK. Occasionally, instead of new operations, speed-up old ones.

Theorem

Let f be a field that augments a RB tree of n nodes, and *suppose the contents of f for node x can be computed in $O(1)$ using only information in node x, left[x] and right[x], including f[left[x]] and f[right[x]]. Then* *we can maintain the values of f in all nodes in T during insertion and deletion without asymptotically affecting $O(\log n)$ performance.*

Proof idea: change in field f at a node x propagates only to ancestors of x in the tree.

- closed interval: $[t_1, t_2]$. Also open, half-open intervals.
- $i = [t_1, t_2]$. $low[i] = t_1$, $high[i] = t_2$.
- i and i' overlap if $i \bigcap i' \neq \emptyset$. That is $low[i] \leq high[i']$ and $low[i'] \leq high[i]$.
- Want: Data structure representing a dynamic set of intervals.
- Must support the following operations:
- $INTERVAL - INSERT(T, x)$: adds element x, whose int field contains an interval.
- $INTERVAL - DELETE(T, x)$: removes element x from T.
- $INTERVAL - SEARCH(T, i)$: return pointer to an element x such that $int[x]$ overlaps i, or nil if no such element found.
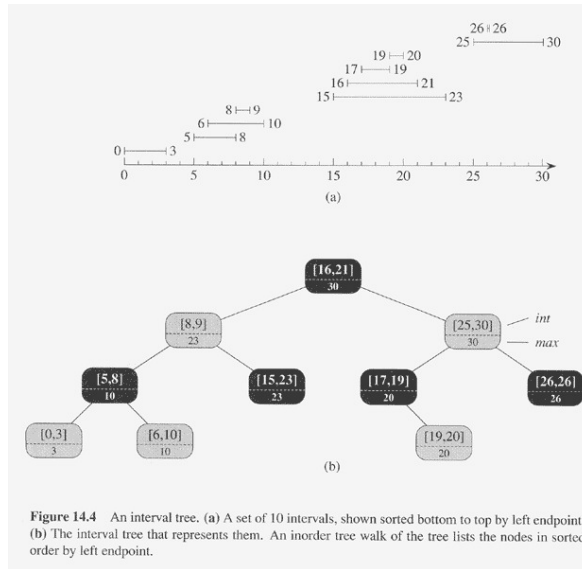
- Any two intervals satisfy interval trichotomy: three alternatives:
  1. i and i' overlap.
  2. i is to the left of i' ($high[i] < low[i']$).
  3. i is to the right of i'. ($low[i] > high[i']$).

(a)

(b)   (c)
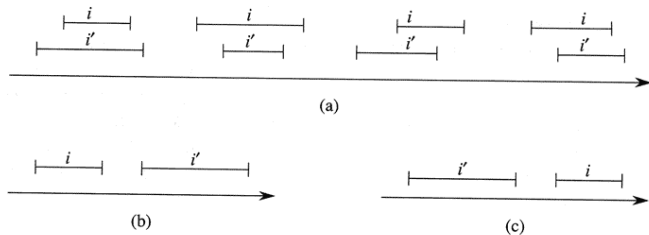
Figure 14.3 The interval trichotomy for two closed intervals i and i'. **(a)** If i and i' overlap, there are four situations; in each, $low[i] \leq high[i']$ and $low[i'] \leq high[i]$. **(b)** The intervals do not overlap, and $high[i] < low[i']$. **(c)** The intervals do not overlap, and $high[i'] < low[i]$.

Interval trees: Implementation

1. Possible clue: intervals (partial) ordering. Might try to modify a total order. Then red-black tree. Each node x stores an interval $int[x]$.
   - $key[x] = low[int[x]]$.
2. Additional info: $max[x]$, the maximum value of any endpoint of an interval stored in the subtree rooted at x.
3. Maintain info: $max[x] = \max(high[int[x]], max[left[x]], max[right[x]])$.
4. By applying previous theorem: insertion/deletion $O(\log n)$ while maintaining $max[x]$.

Figure 14.4 An interval tree. (a) A set of 10 intervals, shown sorted bottom to top by left endpoint. (b) The interval tree that represents them. An inorder tree walk of the tree lists the nodes in sorted order by left endpoint.

INTERVAL-SEARCH

- finds a node in tree T whose interval overlaps interval i, returns sentinel node $nil[T]$ if no overlapping interval found.
- Search starts at the root and proceeds downwards.
- Chooses *left* or *right* subtree based on the maximum element in the left subtree of x.
- If $max[left[x]]$ is $\geq low[i]$ (of course, $left[x] \neq nil[T]$) go left.
- otherwise go right.
- takes $O(\log n)$ time since each basic loop takes $O(1)$ time and the height of the RB tree is $O(\log n)$.

INTERVAL-SEARCH(T, i)
1 $x \leftarrow root[T]$
2 **while** $x \neq nil[T]$ and i does not overlap $int[x]$
3   **do if** $left[x] \neq nil[T]$ and $max[left[x]] \geq low[i]$
4   **then** $x \leftarrow left[x]$
5   **else** $x \leftarrow right[x]$
6 **return** x

- Why is it enough to examine a single path ?
- Idea: search proceeds in a "safe direction".
- INVARIANT: If tree T contains an interval that overlaps i then there is such an interval in the subtree rooted at x.
- Initialization: clearly satisfied, $x = root[T]$.
- Either line 4 or line 5 executed.
- <u>Line 5 executed:</u> because $left[x] = nil[T]$ or $max[left[x]] < low[i]$. The subtree rooted at $left[x]$ does not contain any interval that overlaps i.
- If such an interval is found in T, it must be in $right[x]$.

Correctness of INTERVAL-SEARCH

- <u>Line 4 executed</u>: contrapositive of loop invariant holds.
- If there is no such an interval in the subtree rooted at $left[x]$ then there is no such interval in tree T.
- Since line 4 executed $max[left[x]] \geq low[i]$. There exists i' with $high[i'] = max[left[x]] \geq low[i]$.
- i and i' do not overlap, by assumption. By trichotomy $high[i] < low[i']$.
- i'' interval in $right[x]$. Intervals keyed on the low endpoints.
- $high[i] < low[i'] \leq low[i'']$.
- Conclusion: no interval in $right[x]$ (and thus in T) overlaps i.

Outline:

- Search in secondary storage

- B-Trees
  - ▶ properties
  - ▶ search
  - ▶ insertion

Complexity Model

- Basic assumption so far: *data structures fit completely in main memory (RAM)*
    - ▶ all basic operations have the same cost
        - ▶ even this is a rough approximation, since the main-memory system is not at all "flat"

Complexity Model

- Basic assumption so far: *data structures fit completely in main memory (RAM)*
    - all basic operations have the same cost
        - even this is a rough approximation, since the main-memory system is not at all "flat"

- However, some applications require more storage than what fits in main memory
    - we must use data structures that reside in *secondary storage* (i.e., disk)

Complexity Model

- Basic assumption so far: *data structures fit completely in main memory (RAM)*

  - ▶ all basic operations have the same cost
    - ▶ even this is a rough approximation, since the main-memory system is not at all "flat"

- However, some applications require more storage than what fits in main memory

  - ▶ we must use data structures that reside in *secondary storage* (i.e., disk)
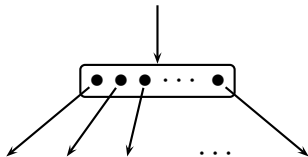
    Disk is 10,000–100,000 times slower than RAM

- In a balanced *binary* tree, n keys require a tree of height $h = \lfloor \log_2 n \rfloor$
    - all the important operations require access to $O(h)$ nodes
    - each one accounting for *one or very few* basic operations

- In a balanced *binary* tree, n keys require a tree of height $h = \lfloor \log_2 n \rfloor$
  - ▶ all the important operations require access to $O(h)$ nodes
  - ▶ each one accounting for *one or very few* basic operations

- **Idea:** store several keys and pointers to children nodes in a single node

Idea

- In a balanced *binary* tree, n keys require a tree of height $h = \lfloor \log_2 n \rfloor$
  - ▶ all the important operations require access to $O(h)$ nodes
  - ▶ each one accounting for *one or very few* basic operations

- **Idea:** store several keys and pointers to children nodes in a single node
  - ▶ in practice we ***increase the degree*** (or *branching factor*) of each node up to $d > 2$, so $h = \lfloor \log_d n \rfloor$
   - ▶ in practice d can be as high as a few thousands

Idea

- In a balanced *binary* tree, n keys require a tree of height $h = \lfloor \log_2 n \rfloor$
  - ▶ all the important operations require access to $O(h)$ nodes
  - ▶ each one accounting for *one or very few* basic operations

- **Idea:** store several keys and pointers to children nodes in a single node
  - ▶ in practice we ***increase the degree*** (or *branching factor*) of each node up to $d > 2$, so $h = \lfloor \log_d n \rfloor$
    - ▶ in practice d can be as high as a few thousands

E.g., if $d = 1000$, then
only three accesses ($h = 2$)
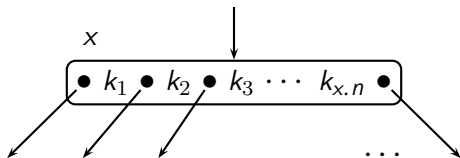cover ***up to one billion keys***

Definition of a B-Tree
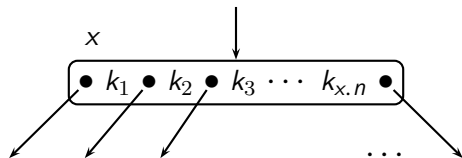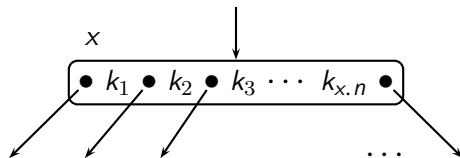
- Every node x has the following fields

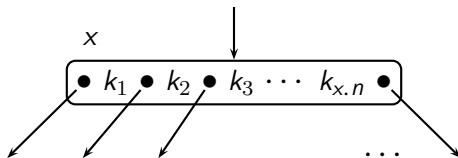   ▶ $x.n$ is the number of keys stored at each node

Definition of a B-Tree

- Every node x has the following fields
    - $x.n$ is the number of keys stored at each node
    - $x.key[1] \leq x.key[2] \leq \ldots x.key[x.n]$ are the $x.n$ keys stored in nondecreasing order
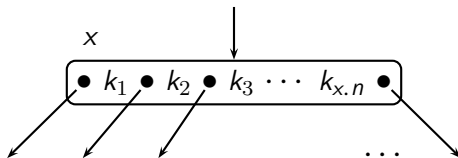
Definition of a B-Tree

- Every node x has the following fields

  - $x.n$ is the number of keys stored at each node
  - $x.key[1] \leq x.key[2] \leq \ldots x.key[x.n]$ are the $x.n$ keys stored in nondecreasing order
  - $x.leaf$ is a Boolean flag that is TRUE if x is a *leaf node* or FALSE if x is an *internal node*
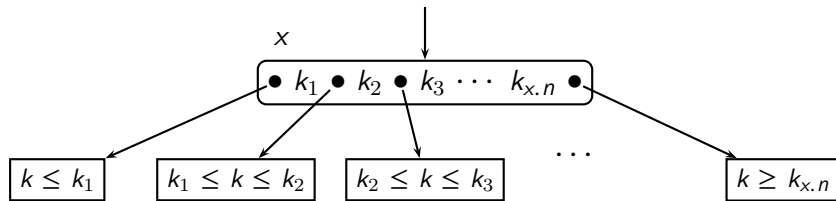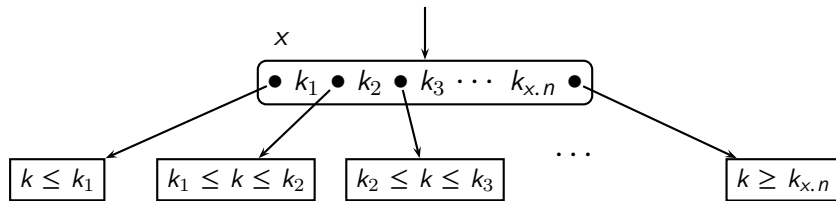
Definition of a B-Tree

- Every node x has the following fields
    - $x.n$ is the number of keys stored at each node
    - $x.key[1] \leq x.key[2] \leq \ldots x.key[x.n]$ are the $x.n$ keys stored in nondecreasing order
    - $x.leaf$ is a Boolean flag that is TRUE if x is a *leaf node* or FALSE if x is an *internal node*
    - $x.c[1], x.c[2], \ldots, x.c[x.n+1]$ are the $x.n+1$ pointers to its children, if x is an *internal node*
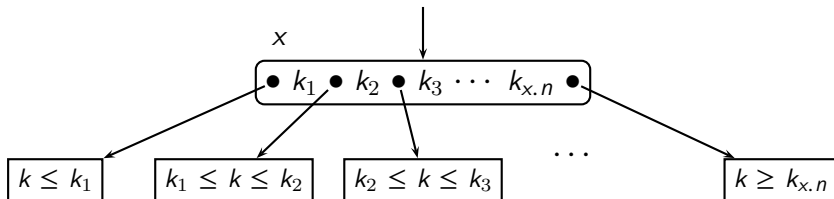
- The keys $x.key[i]$ delimit the ranges of keys stored in each subtree

- The keys $x.key[i]$ delimit the ranges of keys stored in each subtree

  $x.c[1] \longrightarrow$ subtree containing keys $k \leq x.key[1]$

  $x.c[2] \longrightarrow$ subtree containing keys k, $x.key[1] \leq k \leq x.key[2]$

  $x.c[3] \longrightarrow$ subtree containing keys k, $x.key[2] \leq k \leq x.key[3]$

  . . .

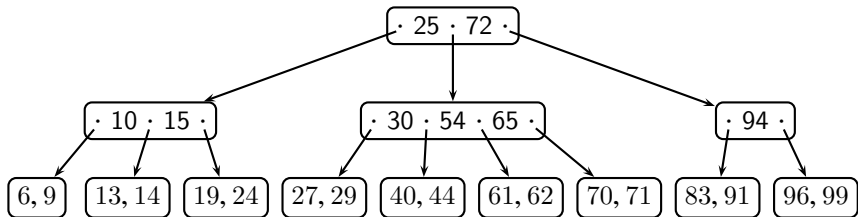 $x.c[x.n + 1] \longrightarrow$ subtree containing keys k, $k \geq x.key[x.n]$

- *All leaves have the same depth*

■ *All leaves have the same depth*

■ Let $t \geq 2$ be the *minimum degree* of the B-tree

  ▶ every node other than the root must have *at least $t - 1$ keys*

  ▶ every node must contain *at most $2t - 1$ keys*

    ▶ a node is *full* when it contains exactly $2t - 1$ keys

    ▶ a full node has $2t$ children

```
B-Tree-Search(x, k) 1   i = 1
                    2   while i ≤ x.n and k > x.key[i]
                    3       i = i + 1
                    4   if i ≤ x.n and k == x.key[i]
                    5       return (x, i)
                    6   if x.leaf
                    7       return NIL
                    8   else Disk-Read(x.c[i])
                    9       return B-Tree-Search(x.c[i], k)
```

- **Theorem:** the height of a B-tree containing $n \geq 1$ keys and with a minimum degree $t \geq 2$ is

$$h \leq \log_t \frac{n+1}{2}$$

- **Theorem:** the height of a B-tree containing $n \geq 1$ keys and with a minimum degree $t \geq 2$ is

$$h \leq \log_t \frac{n+1}{2}$$

Proof:

▶ $n \geq 1$, so the root has at least one key (and therefore two children)

- **Theorem:** the height of a B-tree containing $n \geq 1$ keys and with a minimum degree $t \geq 2$ is

$$h \leq \log_t \frac{n+1}{2}$$

Proof:

- ▶ $n \geq 1$, so the root has at least one key (and therefore two children)
- ▶ every other node has at least t children

- **Theorem:** the height of a B-tree containing $n \geq 1$ keys and with a minimum degree $t \geq 2$ is

$$h \leq \log_t \frac{n+1}{2}$$

Proof:

- ▶ $n \geq 1$, so the root has at least one key (and therefore two children)
- ▶ every other node has at least t children
- ▶ in the worst case, there are two subtrees (of the root) each one containing a total of $(n-1)/2$ keys, and each one consisting of t-degree nodes, with each node containing $t-1$ keys

- **Theorem:** the height of a B-tree containing $n \geq 1$ keys and with a minimum degree $t \geq 2$ is

$$h \leq \log_t \frac{n+1}{2}$$

Proof:

- ▶ $n \geq 1$, so the root has at least one key (and therefore two children)
- ▶ every other node has at least t children
- ▶ in the worst case, there are two subtrees (of the root) each one containing a total of $(n-1)/2$ keys, and each one consisting of t-degree nodes, with each node containing $t-1$ keys
- ▶ each subtree contains $1 + t + t^2 \cdots + t^{h-1}$ nodes, each one containing $t-1$ keys

■ **Theorem:** the height of a B-tree containing $n \geq 1$ keys and with a minimum degree $t \geq 2$ is

$$h \leq \log_t \frac{n+1}{2}$$

Proof:

▶ $n \geq 1$, so the root has at least one key (and therefore two children)

▶ every other node has at least t children

▶ in the worst case, there are two subtrees (of the root) each one containing a total of $(n-1)/2$ keys, and each one consisting of t-degree nodes, with each node containing $t-1$ keys

▶ each subtree contains $1 + t + t^2 \cdots + t^{h-1}$ nodes, each one containing $t-1$ keys, so

$$n \geq 1 + 2(t^h - 1)$$