Algorithms

(Trees)

Pramod Ganapathi
Department of Computer Science
State University of New York at Stony Brook

April 3, 2021

Contents

- General Trees and Binary Trees
- Binary Search Trees
- Balanced Search Trees
  - (2,4)-Trees
  - B Trees
- Tries and Suffix Trees

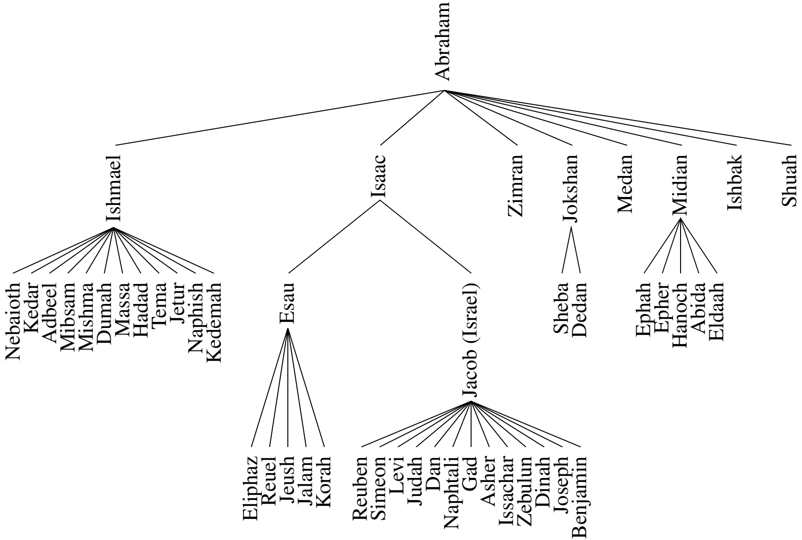
Dictionary operations (for unique keys)

Data structure	Worst case Search	Worst case Insert	Worst case Delete	Average case Search	Average case Insert	Average case Delete
Sorted array	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Unsorted list	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Hashing	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)^*$	$\mathcal{O}(1)^*$	$\mathcal{O}(1)^*$
BST	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Splay tree	$\mathcal{O}(\log n)^*$	$\mathcal{O}(\log n)^*$	$\mathcal{O}(\log n)^*$	$\mathcal{O}(\log n)^*$	$\mathcal{O}(\log n)^*$	$\mathcal{O}(\log n)^*$
Scapegoat tree	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)^*$	$\mathcal{O}(\log n)^*$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
AVL tree	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Red-black tree	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
AA tree	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
(a,b)-tree	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
B-tree	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

* = Amortized

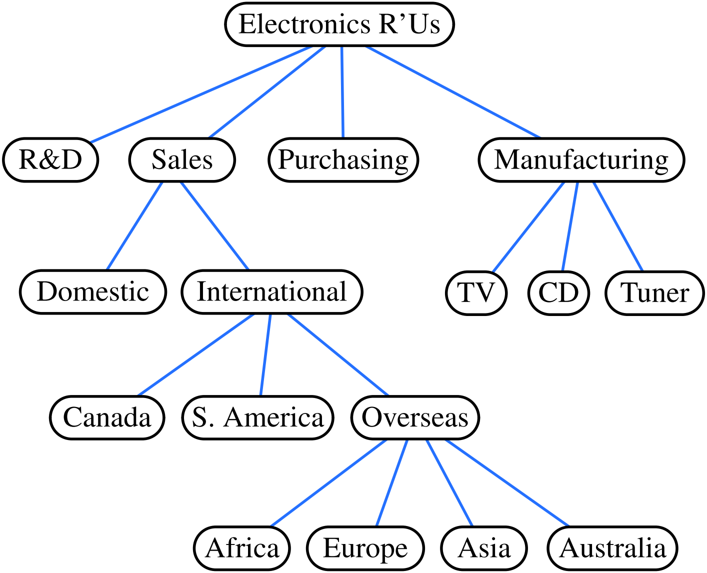
General Trees and Binary Trees

Family tree

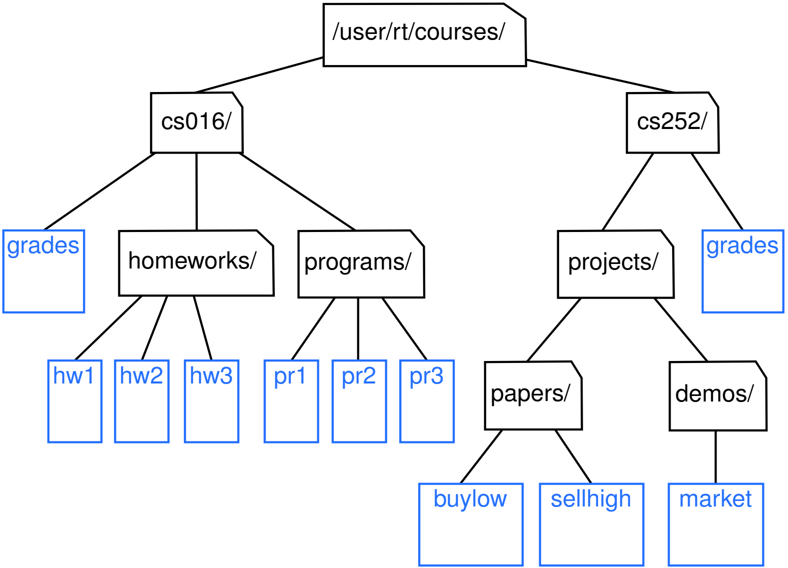


Abraham
Ishmael
Isaac
Zimran
Jokshan
Medan
Midian
Ishbak
Shuah
Nebaioth
Kedar
Adbeel
Mibsam
Mishma
Dumah
Massa
Hadad
Tema
Jetur
Naphish
Kedemah
Esau
Jacob (Israel)
Sheba
Dedan
Ephah
Epher
Hanoch
Abida
Eldaah
Eliphaz
Reuel
Jeush
Jalam
Korah
Reuben
Simeon
Levi
Judah
Dan
Naphtali
Gad
Asher
Issachar
Zebulun
Dinah
Joseph
Benjamin

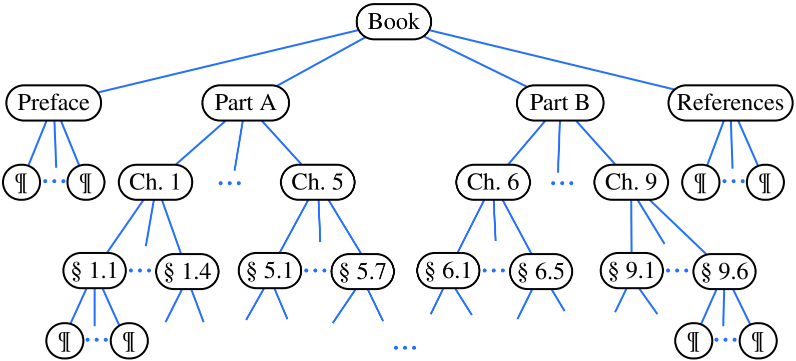
Company organization tree



File system tree



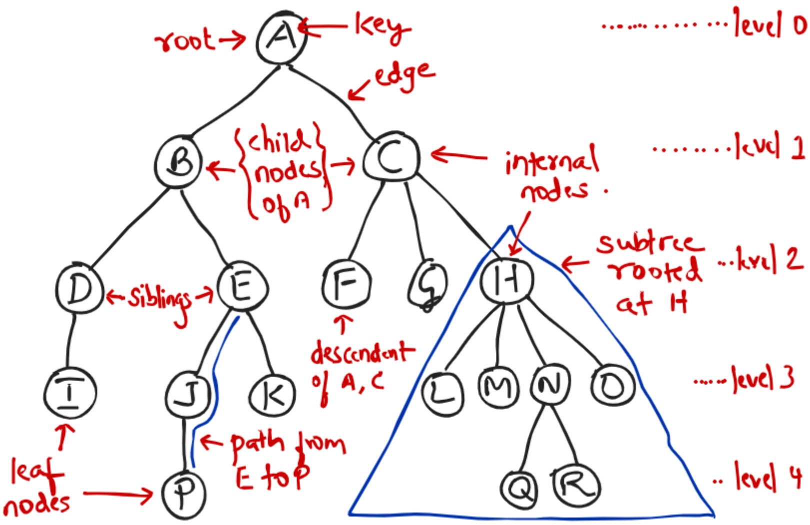
Book organization tree



Terminology

Term	Meaning
Tree	ADT that stores elements hierarchically
Parent node	Immediate previous-level node
Child nodes	Immediate next-level nodes
Root node	Top node of the tree
Sibling nodes	Nodes that are children of the same parent
External nodes	Nodes without children
Internal nodes	Nodes with one or more children
Ancestor node	Parent node or ancestor of parent node
Descendent node	Child node or descendent of child node
Subtree	Tree consisting of the node and its descendants
Edge	Pair of nodes denoting a parent-child relation
Path	Pair of nodes denoting an ancestor-descendant relation
Ordered tree	Tree with a meaningful linear order among child nodes

Terminology



Binary tree

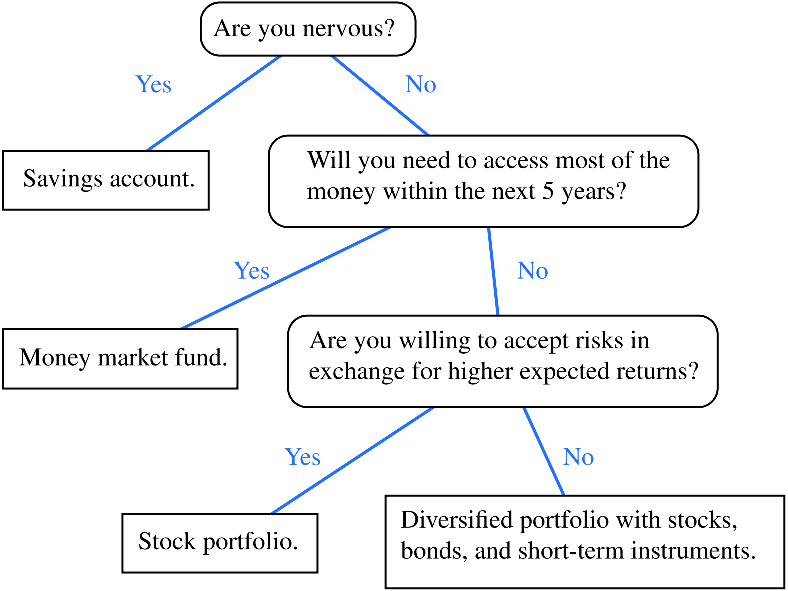
A binary tree is an ordered tree with the following properties:

1. Every node has at most two children.
2. Each child node is labeled as a left child or a right child.
3. A left child precedes a right child in the order of children.

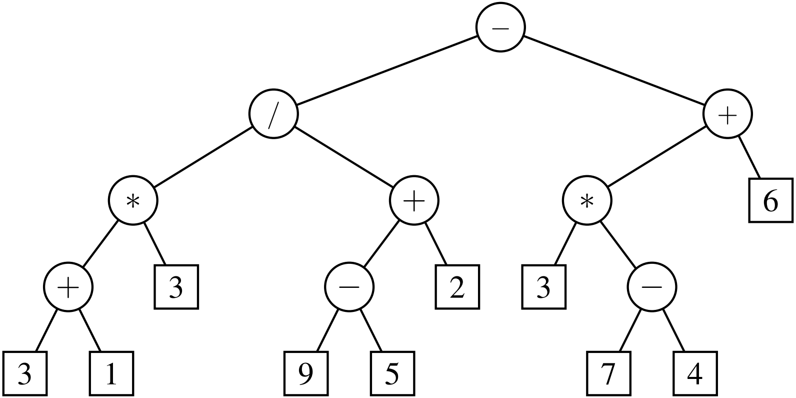
A recursive definition of the binary tree:

- An empty tree.
- A nonempty tree having a root node r, which stores an element, and two binary trees that are respectively the left and right subtrees of r.

Decision tree



Arithmetic expression tree

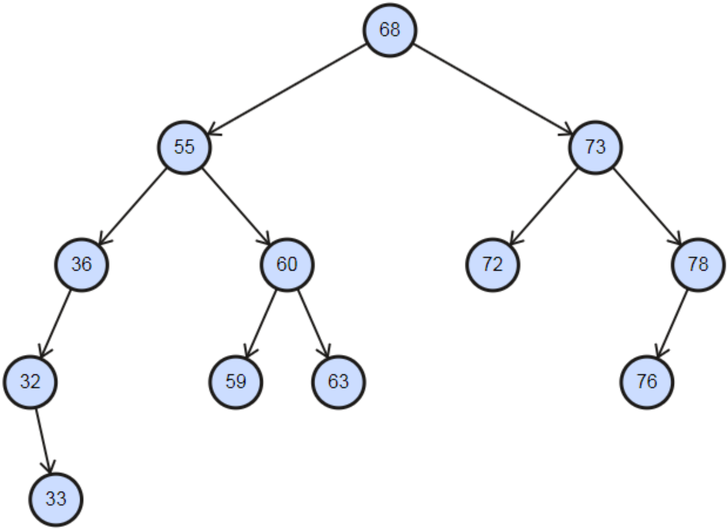


Tree represents $((((3+1)*3)/((9-5)+2))-((3*(7-4))+6))$.

Terminology

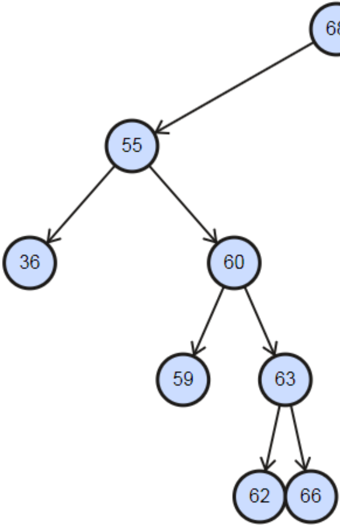
Term	Meaning
Left subtree	Subtree rooted at the left child of an internal node
Right subtree	Subtree rooted at the right child of an internal node
Proper/full tree	A tree in which every node has either 0 or 2 children
Complete tree	Tree in which all except possibly the last level is completely filled and the nodes in the last level are as far left as possible
Perfect tree	Complete tree in which the last level is completely filled

Tree example



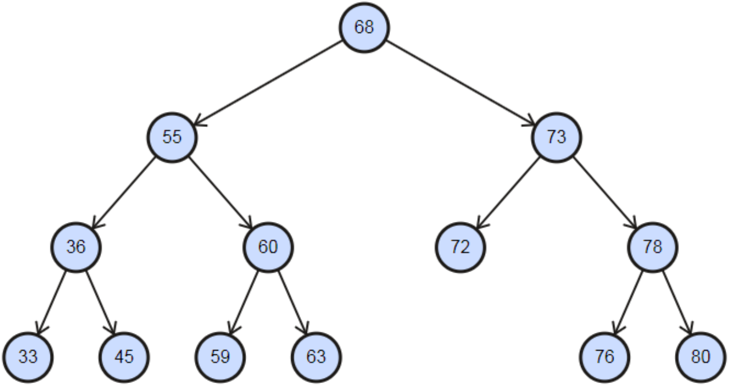
Binary ✓, Proper ✗, Complete ✗, Perfect ✗

Tree example



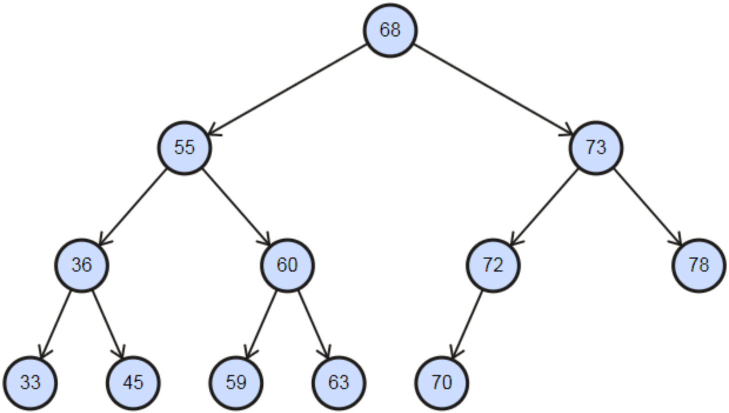
Binary ✓, Proper ✓, Complete ✗, Perfect ✗

Tree example



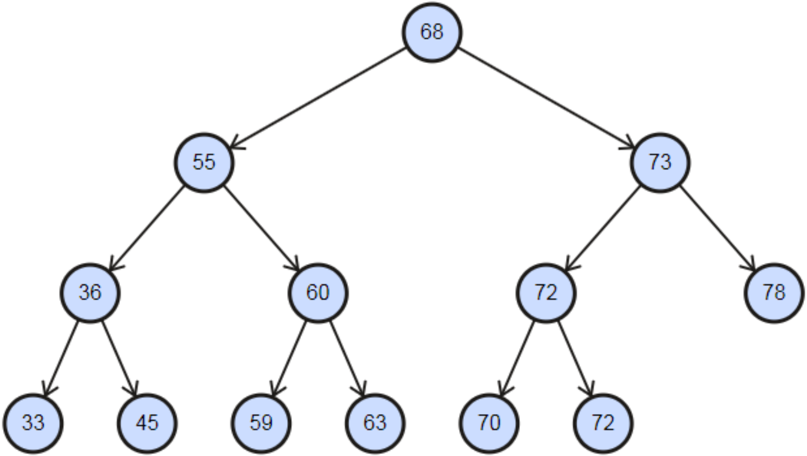
Binary ✓, Proper ✓, Complete ✗, Perfect ✗

Tree example



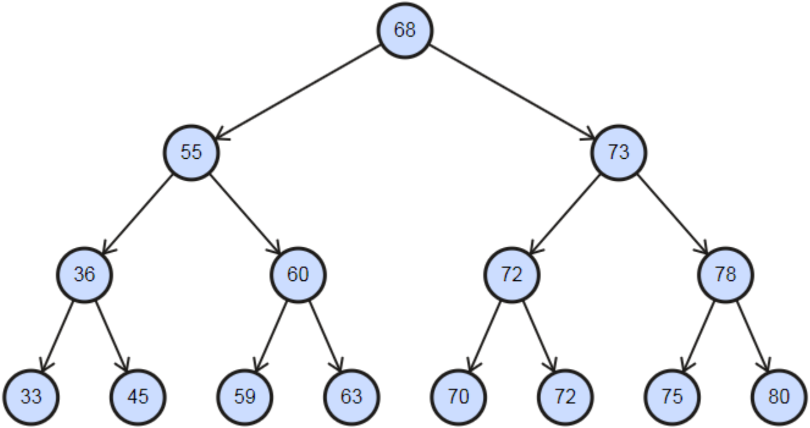
Binary ✓, Proper ✗, Complete ✓, Perfect ✗

Tree example



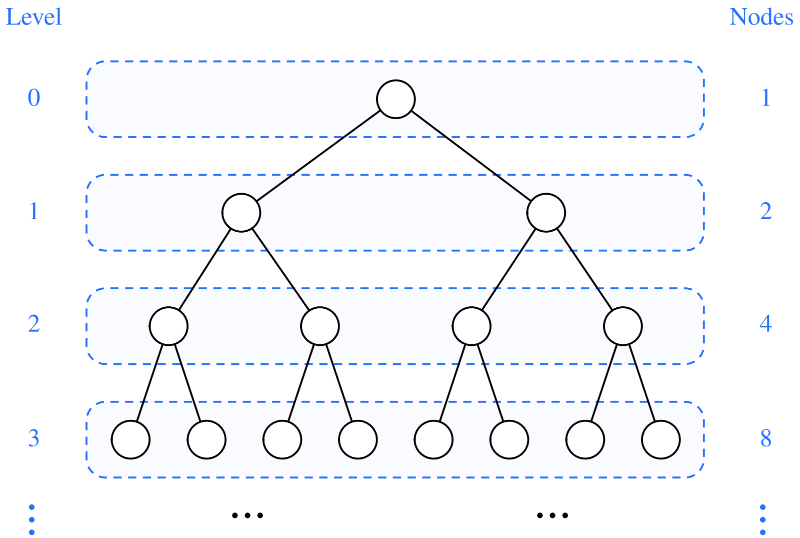
Binary ✓, Proper ✓, Complete ✓, Perfect ✗

Tree example



Binary ✓, Proper ✓, Complete ✓, Perfect ✓

Levels and maximum number of nodes



Properties of binary tree

Let

- $T =$ nonempty binary tree
- $n_{\text{external}} =$ number of external nodes
- $n_{\text{internal}} =$ number of internal nodes
- $n = n_{\text{external}} + n_{\text{internal}}$
- $d_{\text{max}} =$ maximum depth of the tree

Then

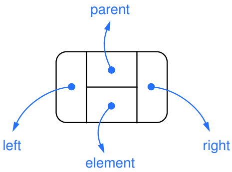
- $dmax + 1 \leq n \leq 2^{d_{\text{max}}+1} - 1$
- $1 \leq n_{\text{external}} \leq 2^{d_{\text{max}}}$
- $d_{\text{max}} \leq n_{\text{internal}} \leq 2^{d_{\text{max}}} - 1$
- $\log(n+1) - 1 \leq d_{\text{max}} \leq n - 1$

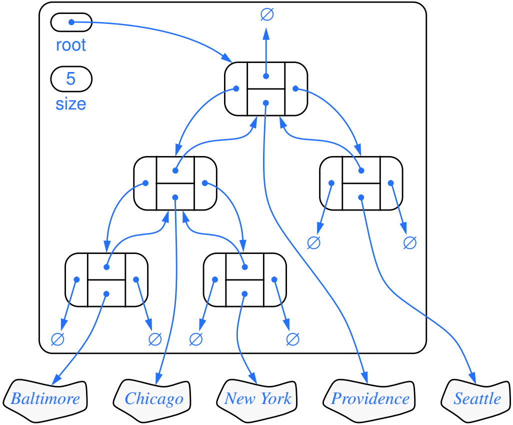
Properties of proper binary tree

If T is a proper nonempty binary tree,

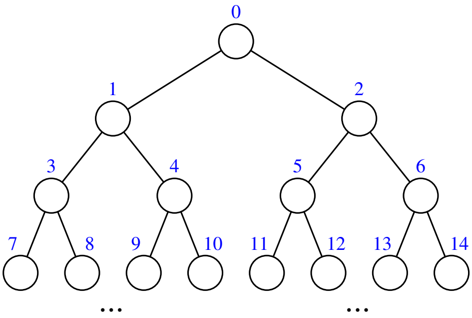
- $2d_{\mathsf{max}} + 1 \le n \le 2^{d_{\mathsf{max}}+1} - 1$
- $d_{\mathsf{max}} + 1 \le n_{\mathsf{external}} \le 2^{d_{\mathsf{max}}}$
- $d_{\mathsf{max}} \le n_{\mathsf{internal}} \le 2^{d_{\mathsf{max}}} - 1$
- $\log(n+1) - 1 \le d_{\mathsf{max}} \le (n-1)/2$
- $n_{\mathsf{external}} = n_{\mathsf{internal}} + 1$

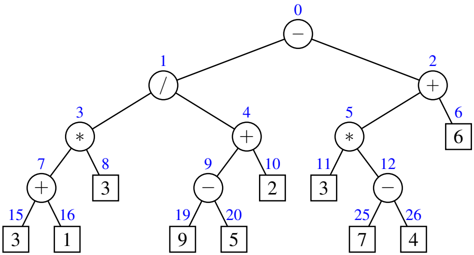
Implementing a binary tree using linked structure



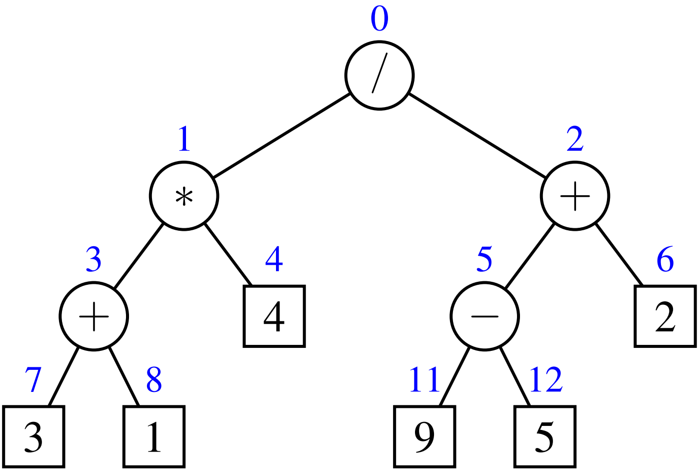


Implementing a binary tree using array





Implementing a binary tree using array



/	*	+	+	4	−	2	3	1			9	5		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

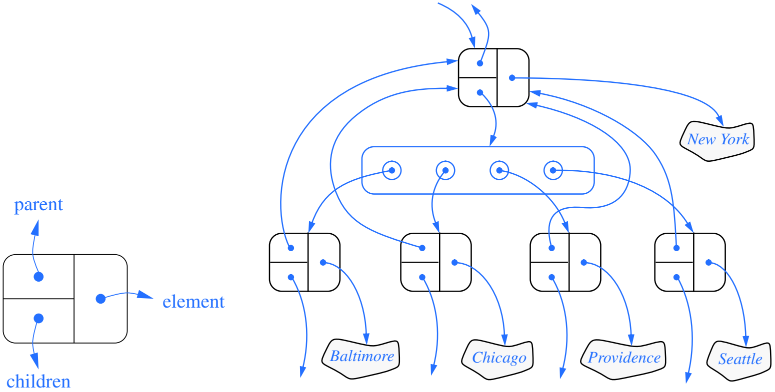
Implementing a binary tree using array

- Level numbering or level ordering
  For every node p of T , let $f(p)$ be the whole number defined as:

$$f(p) = \begin{cases} 0 & \text{if } p \text{ is the root,} \\ 2f(q)+1 & \text{if } p \text{ is the left child of position } q, \\ 2f(q)+2 & \text{if } p \text{ is the right child of position } q. \end{cases}$$

- Then, node p will be stored at index $f(p)$ in the array.
- $0 \leq f(p) \leq 2^n - 1$, where $n =$ number of nodes in T

Implementing a general tree using linked structure



Tree traversals

- A traversal of a tree T is a systematic way of accessing or visiting all the nodes of T.

Traversal	Binary tree?	General tree?
Preorder traversal	✓	✓
Inorder traversal	✓	✗
Postorder traversal	✓	✓
Breadth-first traversal	✓	✓

Preorder/inorder/postorder traversals

$\textsc{PreorderTraversal}(root)$

1. **if** $root \neq null$ **then**
2. $\quad\textsc{Visit}(root)$
3. $\quad\textsc{PreorderTraversal}(root.left)$
4. $\quad\textsc{PreorderTraversal}(root.right)$

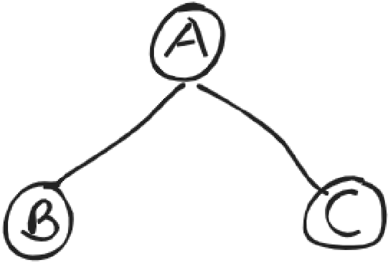
$\textsc{InorderTraversal}(root)$

1. **if** $root \neq null$ **then**
2. $\quad\textsc{InorderTraversal}(root.left)$
3. $\quad\textsc{Visit}(root)$
4. $\quad\textsc{InorderTraversal}(root.right)$

$\textsc{PostorderTraversal}(root)$

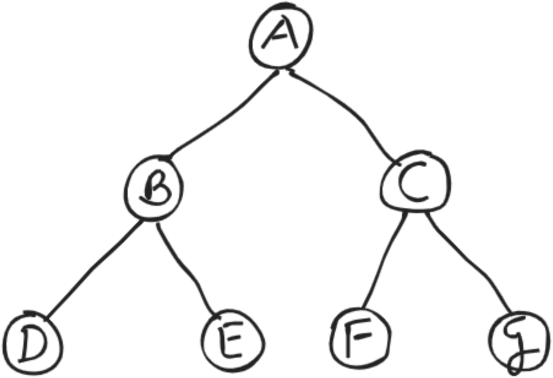
1. **if** $root \neq null$ **then**
2. $\quad\textsc{PostorderTraversal}(root.left)$
3. $\quad\textsc{PostorderTraversal}(root.right)$
4. $\quad\textsc{Visit}(root)$

Preorder/inorder/postorder traversals



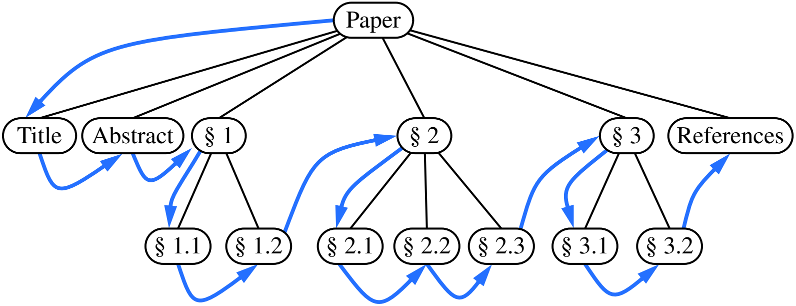
- Preorder traversal = A B C
- Inorder traversal = B A C
- Postorder traversal = B C A

Preorder/inorder/postorder traversals



- Preorder traversal = A [left] [right] = A B D E C F G
- Inorder traversal = [left] A [right] = D B E A F C G
- Postorder traversal = [left] [right] A = D E B F G C A

Preorder traversal

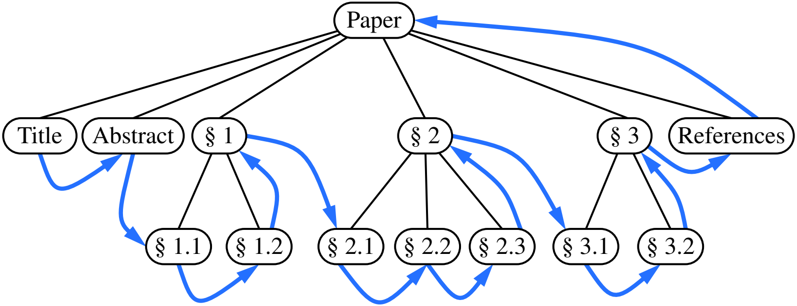


Preorder traversal: Table of contents

```
Paper
Title
Abstract
§1
§1.1
§1.2
§2
§2.1
...
```

```
Paper
  Title
  Abstract
  §1
    §1.1
    §1.2
  §2
    §2.1
    ...
```

Postorder traversal

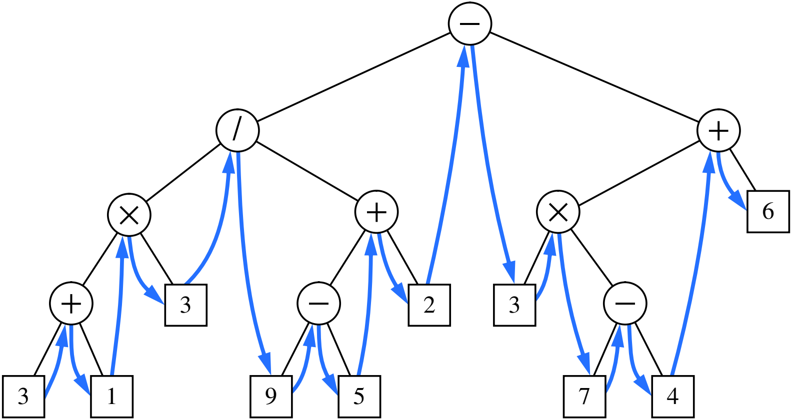


Postorder traversal: Compute disk space

ComputeDiskSpace($root$)

$space \leftarrow root.value$
for each child $child$ of $root$ node **do**
  $space \leftarrow space + \text{ComputeDiskSpace}(root.child)$
return $space$

Inorder traversal: Arithmetic expression



$$(((3+1)\times 3)/((9-5)+2))-((3\times(7-4))+6)$$

Breadth-first traversal

General tree.

BREADTHFIRSTTRAVERSAL()

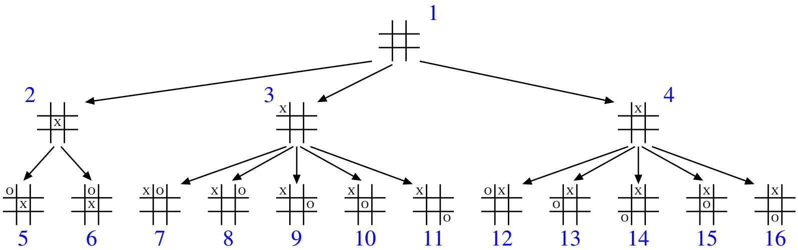
1. Q.enqueue($root$)
2. **while** Q is not empty **do**
3. $curr \leftarrow Q$.dequeue()
4. VISIT($curr$)
5. **for** each child $child$ of $curr$ node **do**
6. Q.enqueue($curr.child$)

Binary tree.

BREADTHFIRSTTRAVERSAL()

1. Q.enqueue($root$)
2. **while** Q is not empty **do**
3. $curr \leftarrow Q$.dequeue()
4. VISIT($curr$)
5. **if** left child exists **then** Q.enqueue($curr.left$)
6. **if** right child exists **then** Q.enqueue($curr.right$)

Breadth-first traversal: Game trees

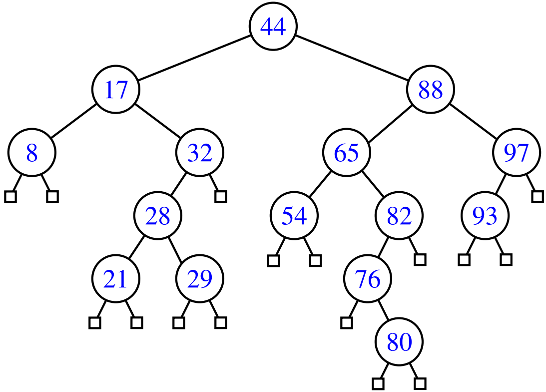


Binary Search Trees (BST)

Binary search tree (BST)

A binary search tree is a proper binary tree T such that, for each internal node p of T:

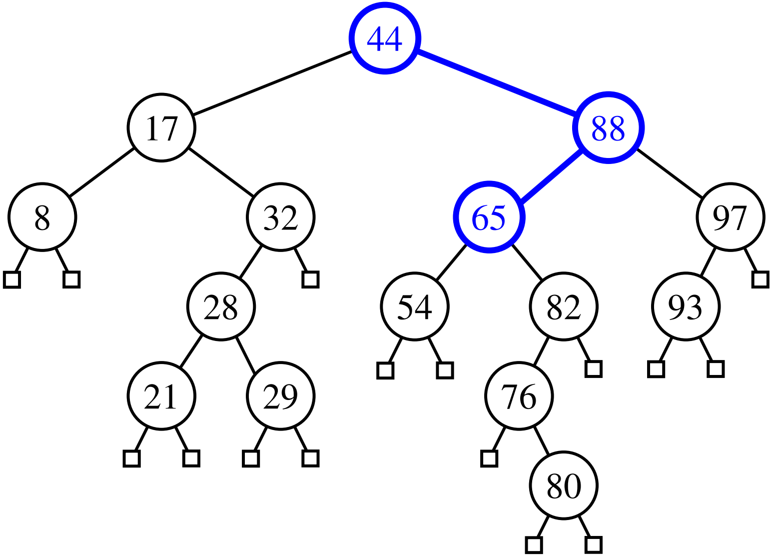
- Node p stores an element, say $p.key$.
- Keys stored in the left subtree of p are less than $p.key$.
- Keys stored in the right subtree of p are greater than $p.key$.



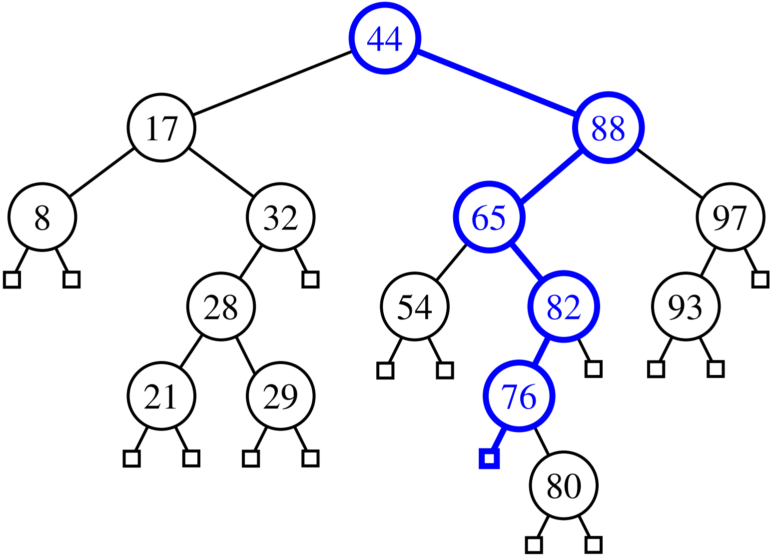
Binary search tree node

```
class Node<T>
{
    T key;
    Node<T> left;
    Node<T> right;

    Node(T item, Node<T> lchild, Node<T> rchild)
    { key = item; left = lchild; right = rchild; }

    Node(T item)
    { this(item, null, null); }
}
```

Search: 65 exists



Search: 68 does not exist



Search: Recursive algorithm

Search$(curr, target)$

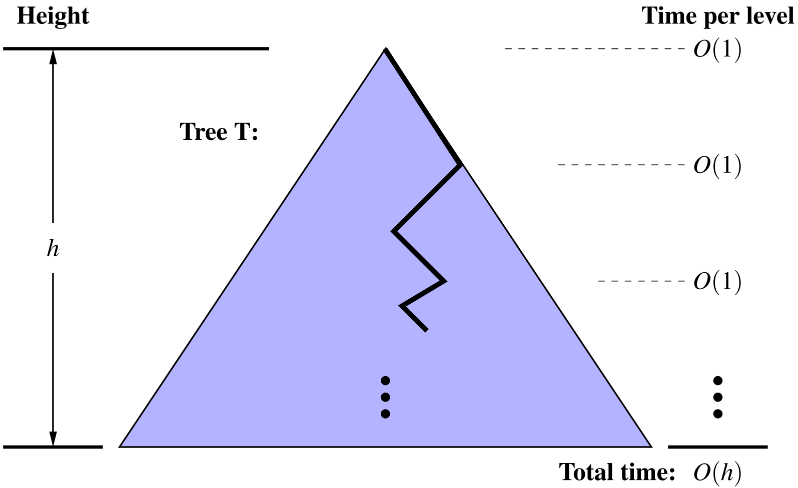
if $curr = null$ **then**
  return $curr$ ▷ unsuccessful search
else if $target < curr.key$ **then**
  return Search$(curr.left, target)$ ▷ recur on left subtree
else if $target > curr.key$ **then**
  return Search$(curr.right, target)$ ▷ recur on right subtree
else if $target = curr.key$ **then**
  return $curr$ ▷ successful search

Search: Iterative algorithm

SEARCH$(curr, target)$

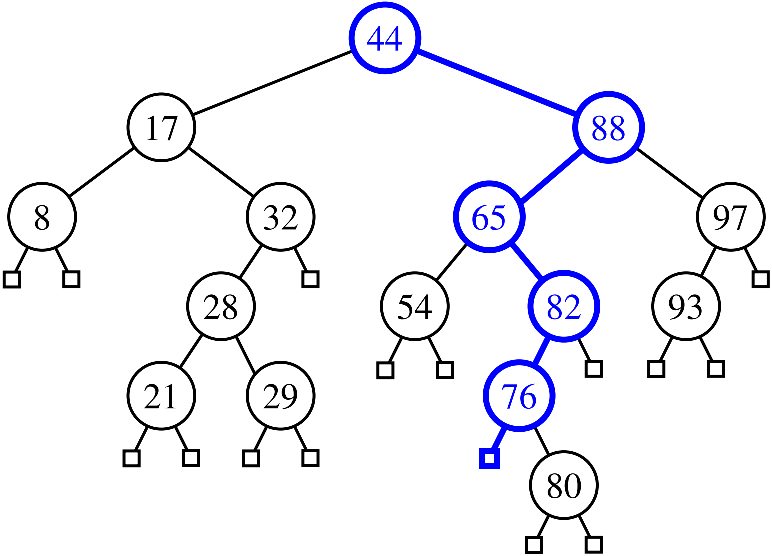
```
while curr ≠ null do
  if target < curr.key then
    curr ← curr.left                        ▷ recur on left subtree
  else if target > curr.key then
    curr ← curr.right                       ▷ recur on right subtree
  else if target = curr.key then
    return curr                             ▷ successful search
return null                                 ▷ unsuccessful search
```

Search: Analysis

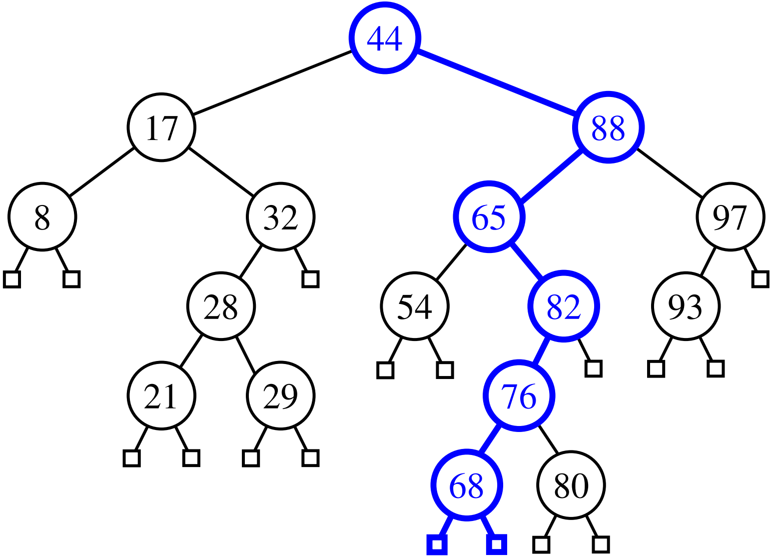


$$\text{Runtime} \in \Theta(h) \in \mathcal{O}(n)$$

Insert 68



Insert 68



Insert: Recursive algorithm

$\textsc{Insert}(curr, item)$

Input: Root of tree and item to be inserted
Output: New root after item insertion

if $curr = null$ **then**
  $curr \leftarrow \textsc{Node}(item)$ ▷ item does not exist

else if $curr \neq null$ **then**
  if $item < curr.key$ **then**
    $curr.left \leftarrow \textsc{Insert}(curr.left, item)$ ▷ recur on left subtree
  else if $item > curr.key$ **then**
    $curr.right \leftarrow \textsc{Insert}(curr.right, item)$ ▷ recur on right subtree
  else if $item = curr.key$ **then**
    do nothing ▷ item exists

return $curr$

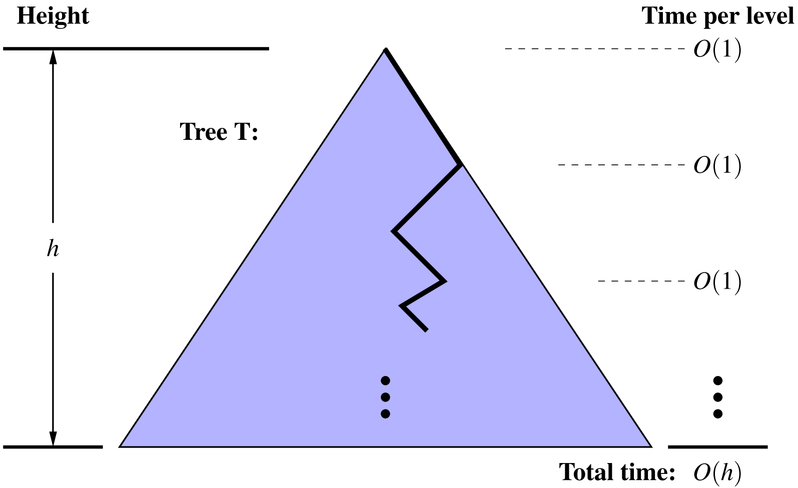
Insert: Iterative algorithm

INSERT(*curr*, *item*)

Input: Root of tree and item to be inserted
Output: Inserted node

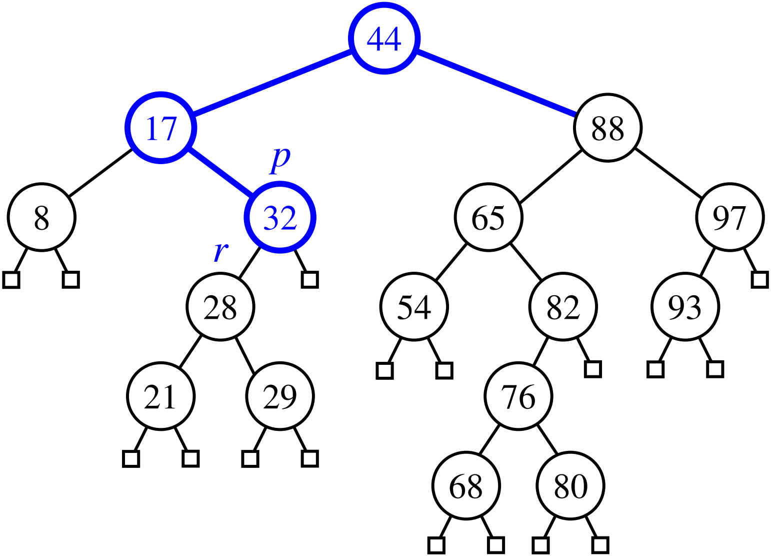
```
prev ← null
while curr ≠ null do
  prev ← curr
  if item < curr.key then
    curr ← curr.left                      ▷ recur on left subtree
  else if item > curr.key then
    curr ← curr.right                     ▷ recur on right subtree
  else if item = curr.key then
    return curr                           ▷ item exists
.....................................................................
curr ← NODE(item)                         ▷ item does not exist
if prev ≠ null then
  if item < prev.key then prev.left ← curr
  if item > prev.key then prev.right ← curr
return curr
```

Insert: Analysis

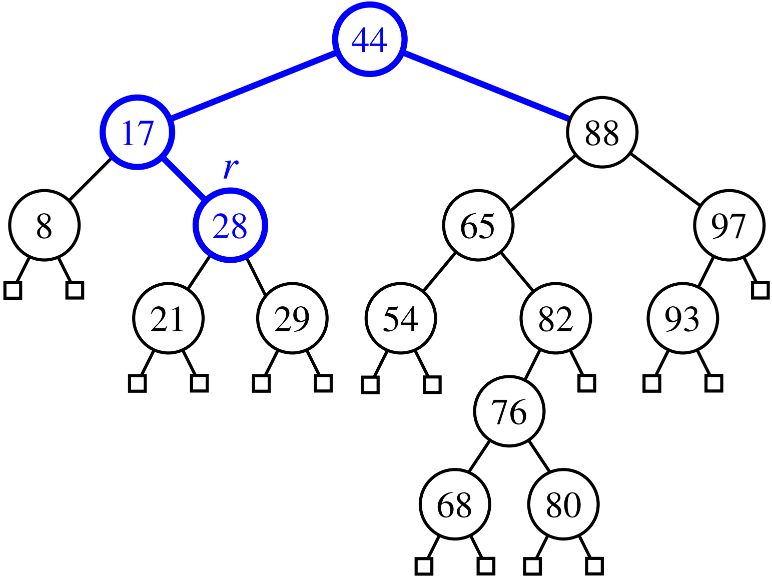


Runtime $\in \Theta(h) \in \mathcal{O}(n)$

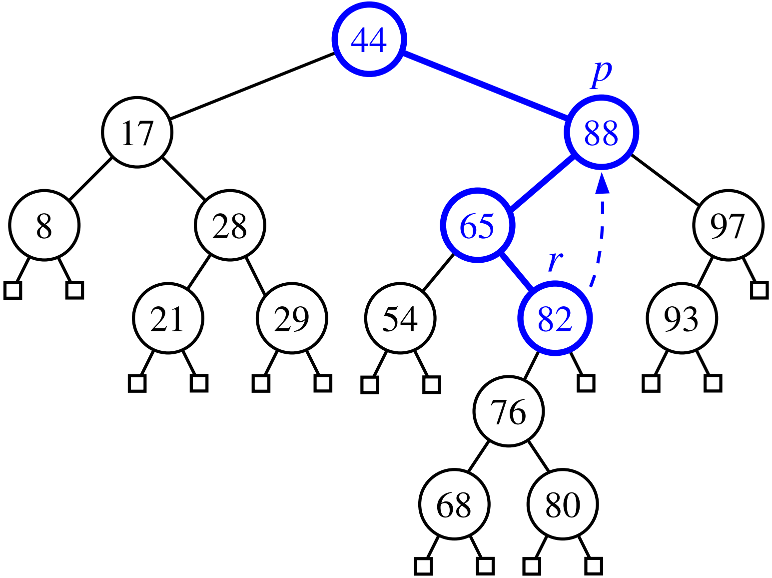
Delete 32: Node 32 has one child



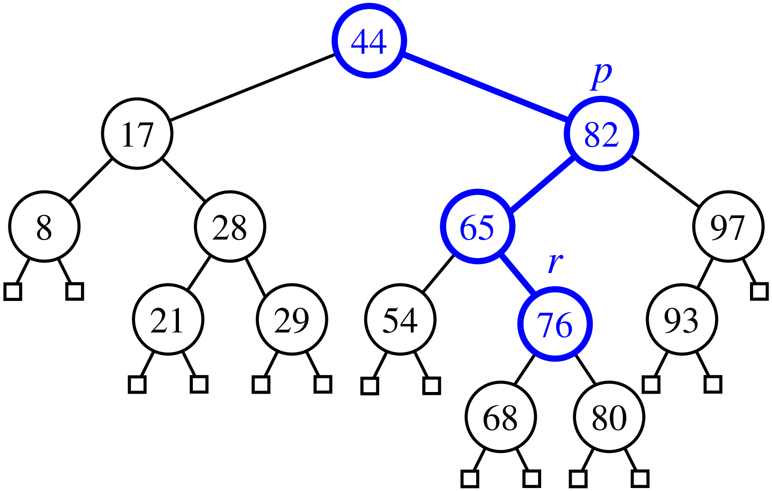
Delete 32: Node 32 has one child



Delete 88: Node 88 has two children



Delete 88: Node 88 has two children



Delete

Deleting a node (with a particular key) has four cases:

1. Node is not found.
   Do nothing.
2. Node is found and it has 0 nonempty children.
   Delete the node.
3. Node is found and it has 1 nonempty child.
   Delete the node.
   Its nonempty child will take the location of the node.
4. Node is found and it has 2 nonempty children.
   Locate the predecessor of the node.
   Predecessor = curr.left.right.right........right
   Predecessor will take the location of the node.
   Predecessor's left child will take the location of the predecessor.
   (Can we use successor instead of predecessor?)

Delete: Recursive algorithm

DELETE(*curr*, *item*)

Input: Root of tree and item to be deleted
Output: New root after item deletion

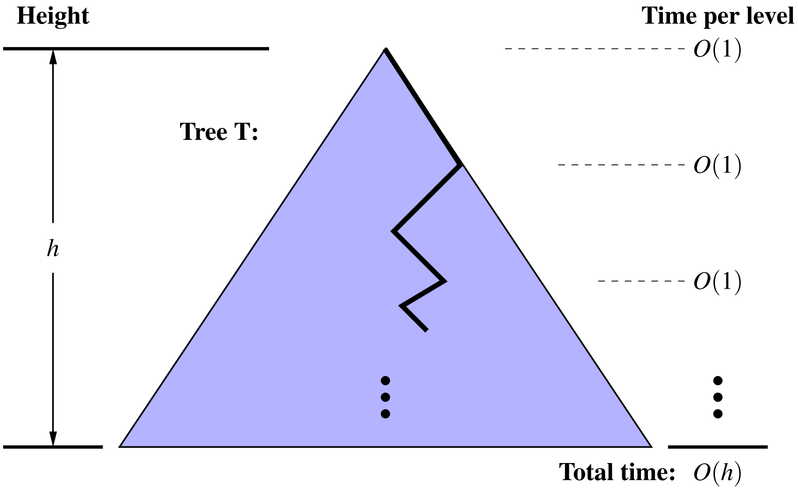
```
if curr = null then
  do nothing                                        ▷ item does not exist
else if item < curr.key then
  curr.left ← DELETE(curr.left, item)               ▷ recur on left
else if item > curr.key then
  curr.right ← DELETE(curr.right, item)             ▷ recur on right
else if item = curr.key then                        ▷ item exists
  ...................................................................
  if curr.left = null then                          ▷ 0 or 1 child
    curr ← curr.right
  else if curr.right = null then                    ▷ 1 child
    curr ← curr.left
  else                                              ▷ 2 children
    curr.key ← FINDMAX(curr.left).key               ▷ find predecessor
    curr.left ← DELETE(curr.left, curr.key)         ▷ delete predecessor
  ...................................................................
return curr
```

Delete: Iterative algorithm

Problem

How do you write an iterative algorithm for deleting an item?

Delete: Analysis



Runtime $\in \Theta(h) \in \mathcal{O}(n)$

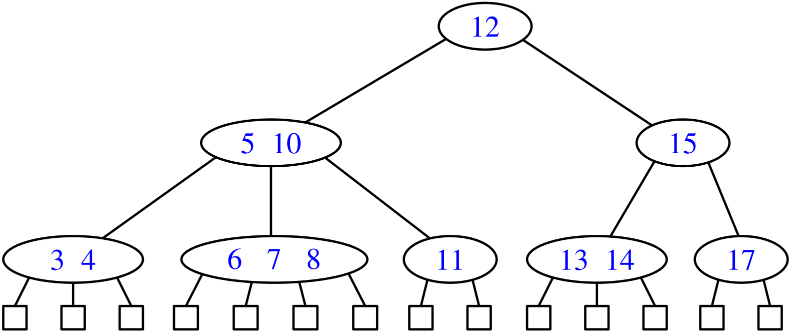
Balanced Search Trees

Balanced search trees: Motivation

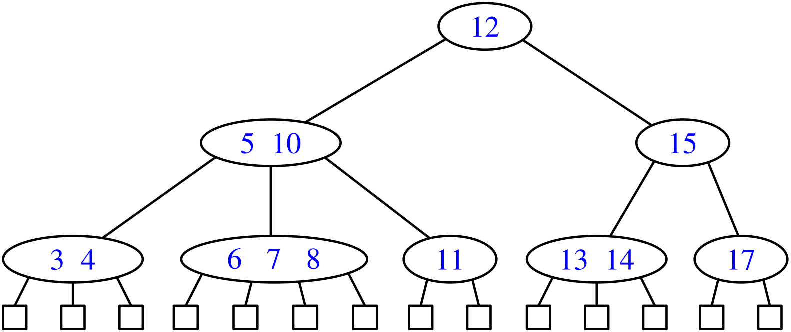
Data structure	Search	Insert	Delete
Binary search tree	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Balanced search tree	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

(2,4)-trees

- A (2,4)-tree or 2-3-4 tree is a balanced search tree.
- A (2,4)-tree satisfies two properties:
  1. Size property. Every non-empty node has 2, 3, or 4 children.
  2. Depth property. All empty nodes have the same depth.



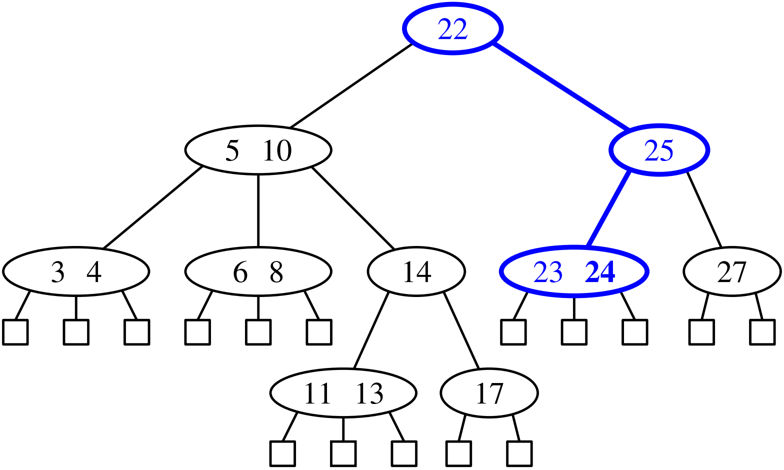
(2,4)-trees



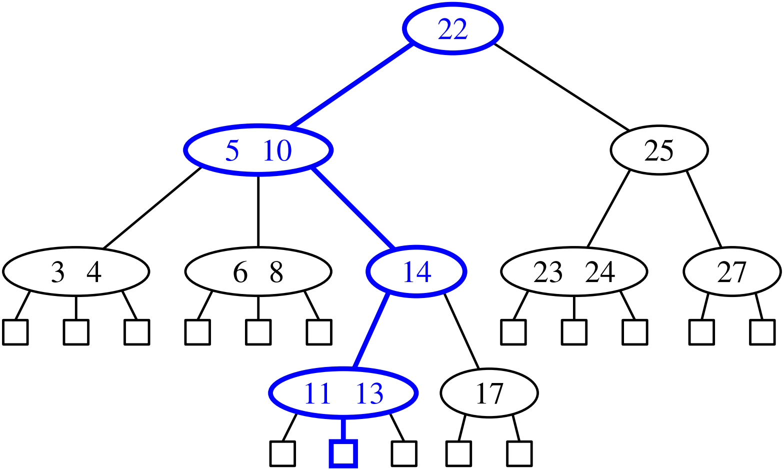
There are three types of non-empty nodes:

- 2-nodes have 2 children and 1 key. e.g.: [11], [12], [15], [17]
- 3-nodes have 3 children and 2 keys. e.g.: [3 4], [5 10], [13 14]
- 4-nodes have 4 children and 3 keys. e.g.: [6 7 8]

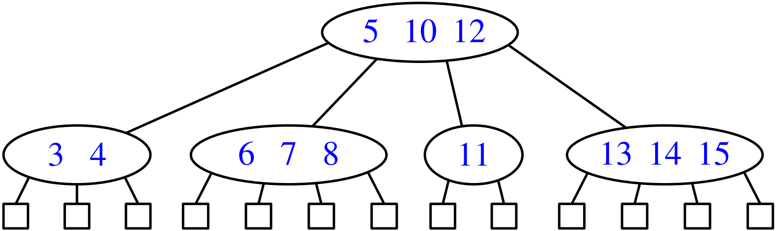
Search: 24 exists



Search: 12 does not exist

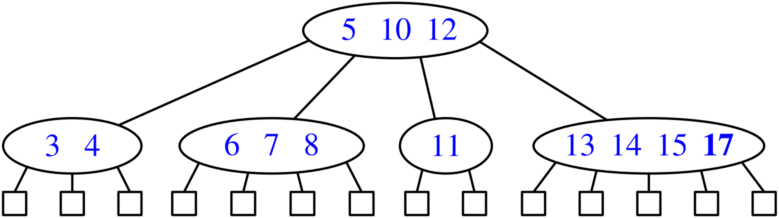


Insert 17



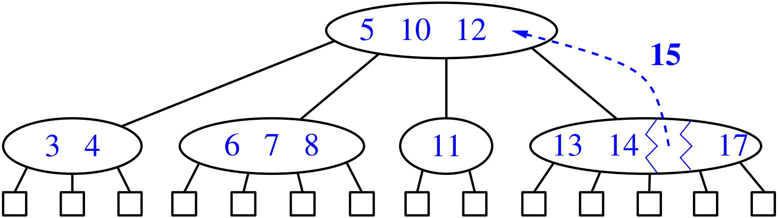
Size and depth properties are satisfied.

Insert 17



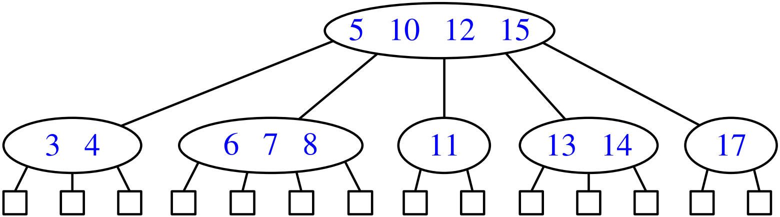
Overflow: Size property is violated at [13 14 15 17].

Insert 17



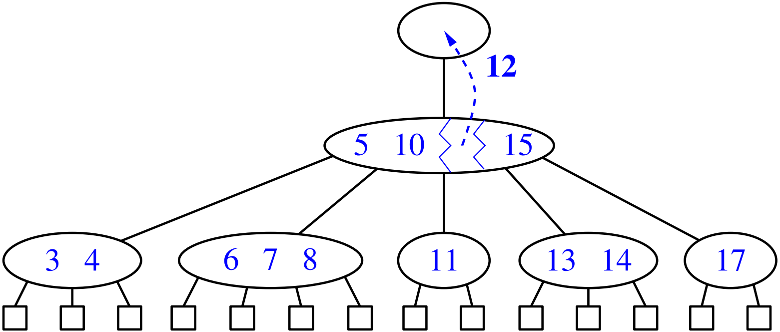
Size property at [13 14 15 17] will be fixed via split operation.

Insert 17



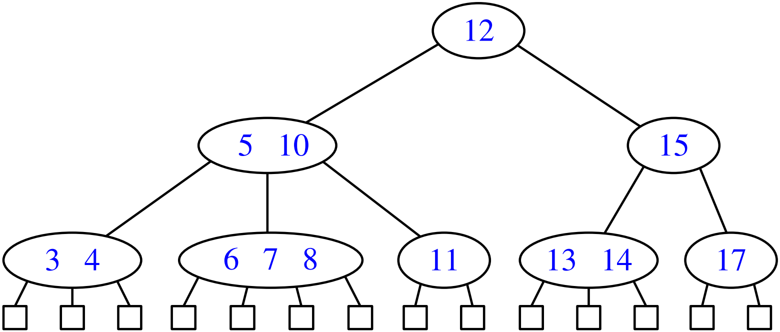
Overflow: Size property is violated at [5 10 12 15].

Insert 17



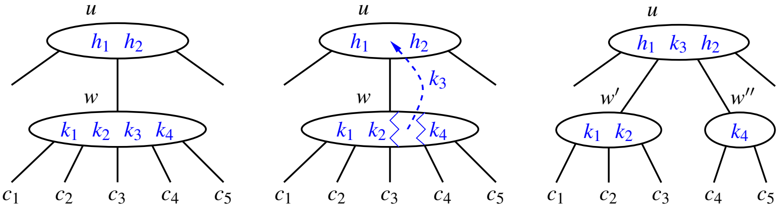
Size property at [5 10 12 15] will be fixed via split operation.

Insert 17

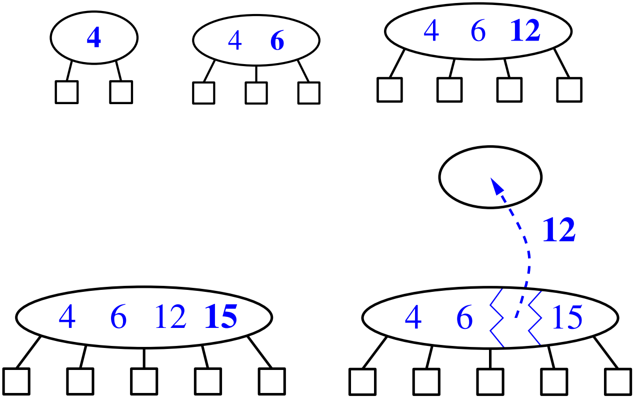


Size and depth properties are satisfied.

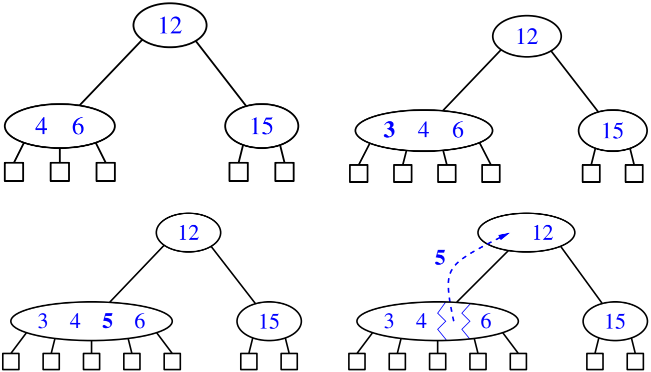
Insert: Node split



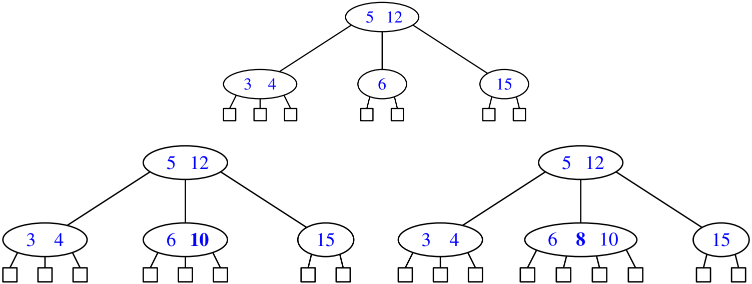
Insert 4, 6, 12, 15



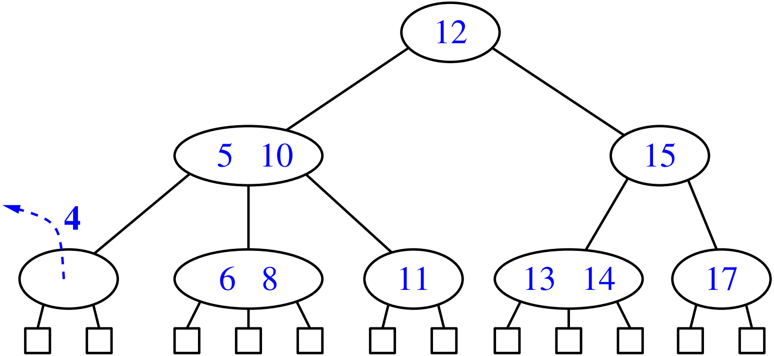
Insert 3, 5



Insert 10, 8

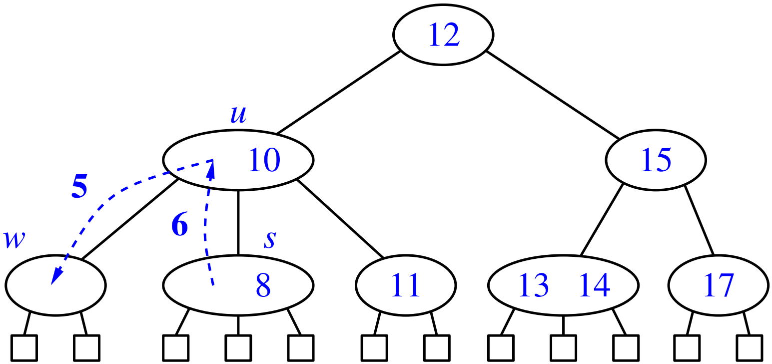


Delete 4



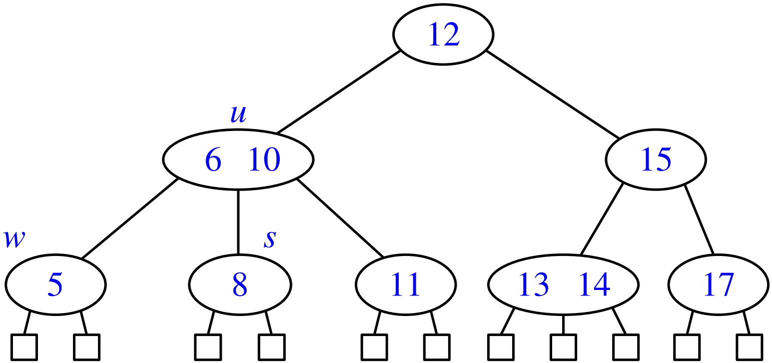
Underflow: Size property is violated is [4].

Delete 4

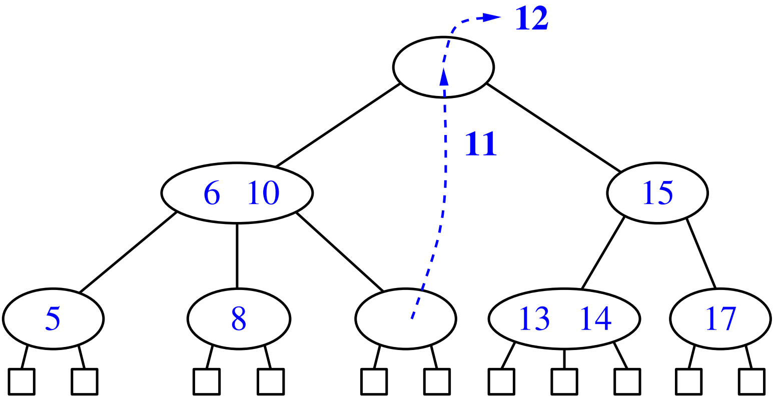


Size property will be fixed via transfer operation.

Delete 12

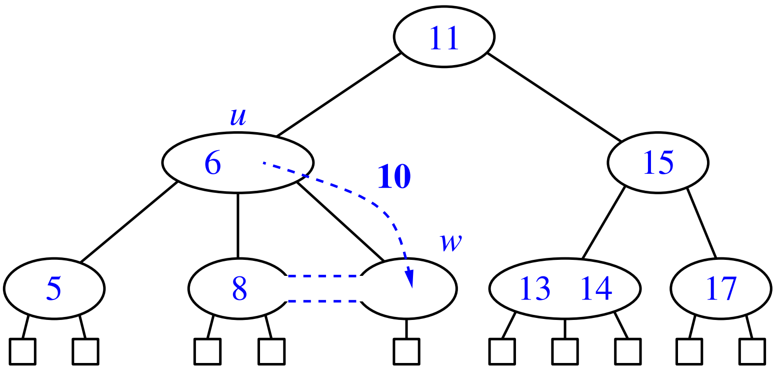


Delete 12



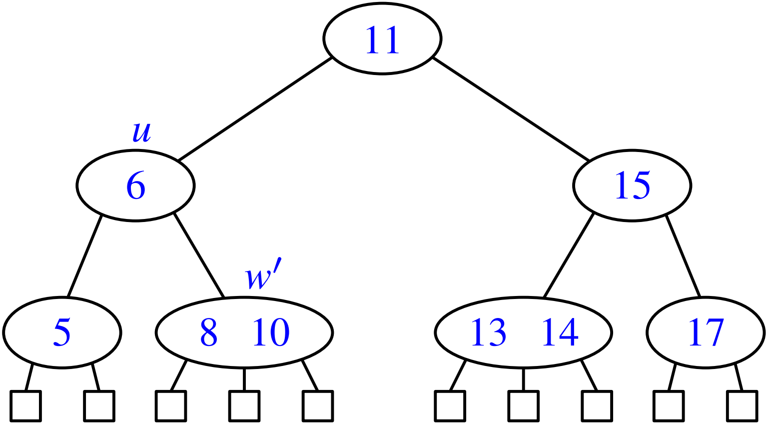
Underflow: Size property is violated is [12], which has non-empty children. It will be fixed via swap with predecessor.
Underflow: Size property is violated is [11].

Delete 12

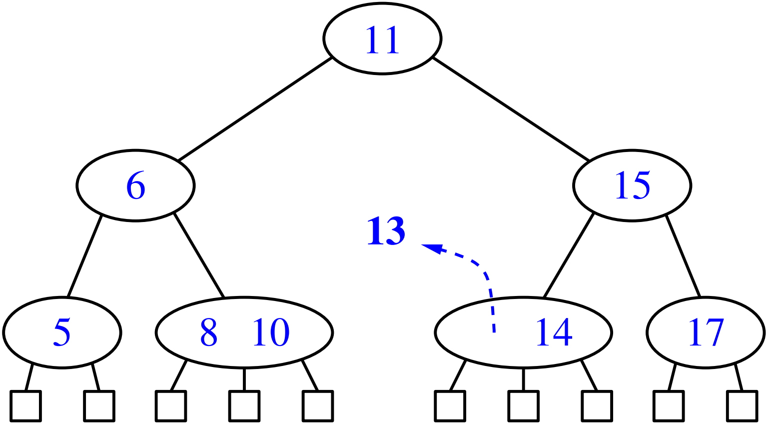


Size property will be fixed via fusion operation.

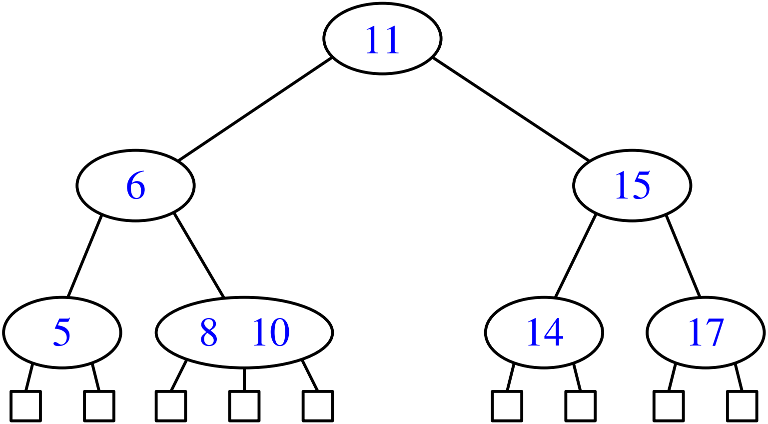
Delete 12



Delete 13



Delete 13



Delete

n_e = node with empty children
$n_{\neq e}$ = node with non-empty children
$s_{3,4}$ = immediate sibling of n_e is a 3-node or a 4-node
s_2 = immediate sibling of n_e is a 2-node
p = parent of n_e

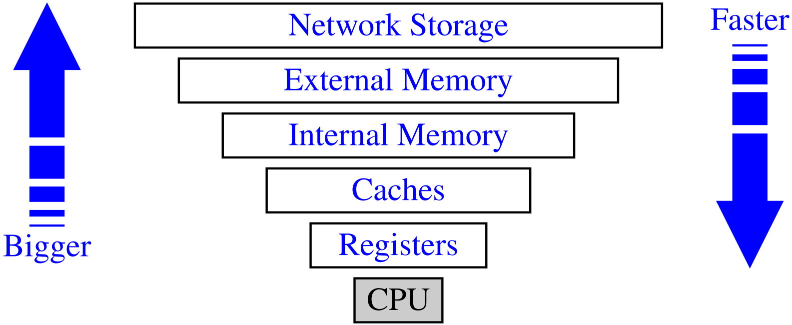
- Deletion of $n_{\neq e}$ can always be reduced to n_e
- Suppose deleted node is:
  1. $n_{\neq e}$.
     Swap with the n_e predecessor
  2. n_e and $s_{3,4}$ exists.
     Transfer a child and key of $s_{3,4}$ to p and a key of p to n_e.
  3. n_e and $s_{3,4}$ does not exist.
     Fuse/merge n_e with s_2 to get n'_e. Move key from p to n'_e.

(2,4)-trees: Complexity

Method	Running time
Search	$\mathcal{O}(\log n)$
Insert	$\mathcal{O}(\log n)$
Delete	$\mathcal{O}(\log n)$

B Trees

Computer memory



Cache-efficient algorithms: Example

Problem

How do you efficiently sort a 1 GB file of natural numbers?

Cache-efficient algorithms: Example

Problem

How do you efficiently sort a 1 GB file of natural numbers?

Workout

Do you want to use quicksort or merge sort, usually implemented in a standard library's sorting algorithm? Your computer program might still take hours to run. Reason? Your algorithm is computation-efficient but not communication-efficient and communication is more expensive than computation.
Reducing communication (via good use of cache) leads to reduced running time. An algorithm that makes good use of cache is called cache-efficient. A cache-efficient sorting algorithm might take just a few minutes to sort a 1 GB file of numbers.
Example: External-memory merge sort.

Cache data locality

An algorithm must have the following two features in order to make good use of cache.

1. Spatial data locality
2. Temporal data locality

Spatial data locality

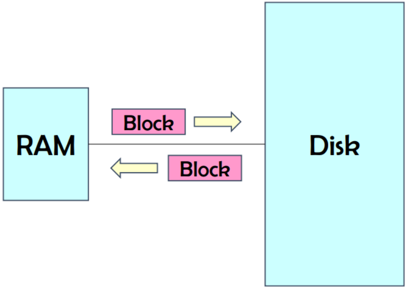
- Meaning?
  Whenever a cache block is brought into the cache, it contains as much useful data as possible.
- How to exploit?
  Group data in blocks (or pages). Move data in blocks.

Temporal data locality

- Meaning?
  Whenever a cache block is brought into the cache, as much useful work as possible is performed on this data before removing the block from the cache.
- Necessary condition?
  Total computations is asymptotically greater than space
  i.e., $T(n) \in \omega\left(S(n)\right)$
- How to exploit?
  Design recursive divide-and-conquer algorithms

Cache complexity

- Cache complexity is the asymptotic number of cache misses or page faults incurred by an algorithm.
- Cache-efficient algorithms incur fewer cache misses.
- Cache-efficient algorithms try to exploit both spatial and temporal data locality.
- Terminology: B = data block size, M = cache size



Cache-efficient algorithms

Problem	Cache-inefficient algo	Cache-efficient algo
Sorting	Merge sort $\mathcal{O}(n \log n)$	Ext-memory merge sort $\mathcal{O}\left(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{B}\right)$
Balanced tree	(2,4)-tree $\mathcal{O}(\log n)$	B tree $\mathcal{O}(\log_B n)$
Matrix product	Iterative $\mathcal{O}(n^3)$	Recursive D&C $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$

(a, b)-trees

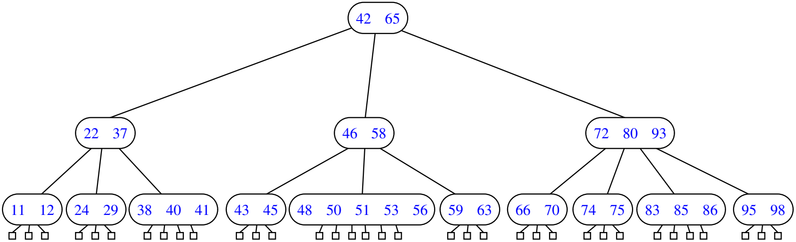
- (a, b)-tree is a straightforward generalization of (2,4)-tree in which the complexities depend on a and b
- By choosing proper values for a and b, we get a balanced search tree that has excellent external-memory performance
- (a, b)-tree is a multiway search tree such that each node has between a and b children and stores between $a - 1$ and $b - 1$ entries

(a, b)-trees

- An (a, b)-tree is a balanced multiway search tree.
- An (a, b)-tree satisfies three properties:
  1. $2 \leq a \leq (b + 1)/2$
  2. Size property. Every non-empty node has children in the range $[a, b]$.
  3. Depth property. All empty nodes have the same depth.

B trees

- B tree of order d is an (a, b) tree with $a = \lceil d/2 \rceil$ and $b = d$.
- B trees are analyzed for cache complexity.
- B trees are cache-efficient, when $d = B$, as they exploit spatial data locality.



B trees: Complexity

Method	(2,4)-tree Communication	(2,4)-tree Computation	B tree Communication	B tree Computation
Search	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log_B n)$	$\mathcal{O}(\log n)$
Insert	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log_B n)$	$\mathcal{O}(\log n)$
Delete	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log_B n)$	$\mathcal{O}(\log n)$

- B trees (and variants such as B+ trees, B* trees, B# trees) are used for file systems and databases.
  Microsoft: NTFS
  Mac: HFS, HFS+
  Linux: BTRFS, EXT4, JFS2
  Databases: Oracle, DB2, Ingres, SQL, PostgreSQL