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)

<table>
<thead>
<tr>
<th>Data structure</th>
<th>Worst case</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Average case</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Search</td>
<td>Insert</td>
<td>Delete</td>
<td>Search</td>
<td>Insert</td>
<td>Delete</td>
<td>Search</td>
<td>Insert</td>
<td>Delete</td>
</tr>
<tr>
<td>Sorted array</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unsorted list</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hashing</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(1)^*$</td>
<td>$O(1)^*$</td>
<td>$O(1)^*$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BST</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Splay tree</td>
<td>$O(\log n)^*$</td>
<td>$O(\log n)^*$</td>
<td>$O(\log n)^*$</td>
<td>$O(\log n)^*$</td>
<td>$O(\log n)^*$</td>
<td>$O(\log n)^*$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scapegoat tree</td>
<td>$O(\log n)$</td>
<td>$O(\log n)^*$</td>
<td>$O(\log n)^*$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVL tree</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Red-black tree</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AA tree</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(a,b)$-tree</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B-tree</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$* = \text{Amortized}$
General Trees and Binary Trees
<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree</td>
<td>ADT that stores elements hierarchically</td>
</tr>
<tr>
<td>Parent node</td>
<td>Immediate previous-level node</td>
</tr>
<tr>
<td>Child nodes</td>
<td>Immediate next-level nodes</td>
</tr>
<tr>
<td>Root node</td>
<td>Top node of the tree</td>
</tr>
<tr>
<td>Sibling nodes</td>
<td>Nodes that are children of the same parent</td>
</tr>
<tr>
<td>External nodes</td>
<td>Nodes without children</td>
</tr>
<tr>
<td>Internal nodes</td>
<td>Nodes with one or more children</td>
</tr>
<tr>
<td>Ancestor node</td>
<td>Parent node or ancestor of parent node</td>
</tr>
<tr>
<td>Descendent node</td>
<td>Child node or descendent of child node</td>
</tr>
<tr>
<td>Subtree</td>
<td>Tree consisting of the node and its descendants</td>
</tr>
<tr>
<td>Edge</td>
<td>Pair of nodes denoting a parent-child relation</td>
</tr>
<tr>
<td>Path</td>
<td>Pair of nodes denoting an ancestor-descendant relation</td>
</tr>
<tr>
<td>Ordered tree</td>
<td>Tree with a meaningful linear order among child nodes</td>
</tr>
</tbody>
</table>
 Terminology

- **Root:** A
- **Key:** A
- **Edge:** B -> A
- **Child nodes of A:** B, C
- **Internal nodes:** B, C
- **Siblings:** D, E, F, G
- **Leaf nodes:** I, J, K, L, M, N, O, P
- **Path from E to P:** E -> F -> G -> P
- **Descendant of A, C:** I, J, K
- **Subtree rooted at H:** H -> L, M, N, O, Q, R
- **Level 0:** A
- **Level 1:** B, C
- **Level 2:** D, E, F, G, H
- **Level 3:** I, J, K, L, M, N, O
- **Level 4:** P, Q, R
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

- Are you nervous?
  - Yes: Savings account.
  - No: Will you need to access most of the money within the next 5 years?
    - Yes: Money market fund.
    - No: Are you willing to accept risks in exchange for higher expected returns?
      - Yes: Stock portfolio.
      - No: Diversified portfolio with stocks, bonds, and short-term instruments.
Tree represents \(((3 + 1) \times 3)/(9 - 5 + 2)) - (3 \times (7 - 4)) + 6\).
<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left subtree</td>
<td>Subtree rooted at the left child of an internal node</td>
</tr>
<tr>
<td>Right subtree</td>
<td>Subtree rooted at the right child of an internal node</td>
</tr>
<tr>
<td>Proper/full tree</td>
<td>A tree in which every node has either 0 or 2 children</td>
</tr>
<tr>
<td>Complete tree</td>
<td>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</td>
</tr>
<tr>
<td>Perfect tree</td>
<td>Complete tree in which the last level is completely filled</td>
</tr>
</tbody>
</table>
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

- \( d_{\text{max}} + 1 \leq n \leq 2^{d_{\text{max}}+1} - 1 \)
- \( 1 \leq n_{\text{external}} \leq 2^{d_{\text{max}}} \)
- \( d_{\text{max}} \leq n_{\text{internal}} \leq 2^{d_{\text{max}}} - 1 \)
- \( \log(n + 1) - 1 \leq d_{\text{max}} \leq n - 1 \)
If $T$ is a proper nonempty binary tree,

- $2d_{\text{max}} + 1 \leq n \leq 2^{d_{\text{max}}+1} - 1$
- $d_{\text{max}} + 1 \leq n_{\text{external}} \leq 2^{d_{\text{max}}}$
- $d_{\text{max}} \leq n_{\text{internal}} \leq 2^{d_{\text{max}}} - 1$
- $\log(n + 1) - 1 \leq d_{\text{max}} \leq (n - 1)/2$
- $n_{\text{external}} = n_{\text{internal}} + 1$
Implementing a binary tree using linked structure
Implementing a binary tree using array
Implementing a binary tree using array

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

```
          0
           |
            /
            |
            |
         1   2
       /     /
     3     5
    / 
   4   6
```

```
  7
 / 
3 8
```

```
3 1
```

```
4
```

```
9 5
```

```
11 12
```

```
2
```
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

- parent
- element
- children

- Baltimore
- Chicago
- Providence
- Seattle

- New York
Tree traversals

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

<table>
<thead>
<tr>
<th>Traversal</th>
<th>Binary tree?</th>
<th>General tree?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preorder traversal</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Inorder traversal</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Postorder traversal</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Breadth-first traversal</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
### PreorderTraversal(root)

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

### InorderTraversal(root)

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

### PostorderTraversal(root)

1. **if** root $\neq$ null **then**
2. PostorderTraversal(root.left)
3. PostorderTraversal(root.right)
4. 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
- Postorder traversal = [left] [right] A = D E B F G C A
Preorder traversal
### Preorder traversal: Table of contents

<table>
<thead>
<tr>
<th>Paper</th>
<th>Paper</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Title</td>
</tr>
<tr>
<td>Abstract</td>
<td>Abstract</td>
</tr>
<tr>
<td>§1</td>
<td>§1</td>
</tr>
<tr>
<td>§1.1</td>
<td>§1.1</td>
</tr>
<tr>
<td>§1.2</td>
<td>§1.2</td>
</tr>
<tr>
<td>§2</td>
<td>§2</td>
</tr>
<tr>
<td>§2.1</td>
<td>§2.1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Postorder traversal: Compute disk space

\begin{tabular}{|l|}
\hline
\textbf{COMPUTE_DISK_SPACE}(root) \\
\hline
1. \textit{space} $\leftarrow$ \textit{root.value} \\
2. \textbf{for} each child child of root node \textbf{do} \\
3. \textit{space} $\leftarrow$ \textit{space} + \text{COMPUTE_DISK_SPACE}(\textit{root.child}) \\
4. \textbf{return} \textit{space} \\
\hline
\end{tabular}
(((3 + 1) × 3)/((9 − 5) + 2)) − ((3 × (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)
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 $p.key$. 
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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Search</strong>(curr, target)</td>
<td></td>
</tr>
<tr>
<td>1. <strong>if</strong> curr = null <strong>then</strong></td>
<td></td>
</tr>
<tr>
<td>2. <strong>return</strong> curr</td>
<td>▶ unsuccessful search</td>
</tr>
<tr>
<td>3. <strong>else if</strong> target &lt; curr.key <strong>then</strong></td>
<td></td>
</tr>
<tr>
<td>4. <strong>return</strong> Search(curr.left, target)</td>
<td>▶ recur on left subtree</td>
</tr>
<tr>
<td>5. <strong>else if</strong> target &gt; curr.key <strong>then</strong></td>
<td></td>
</tr>
<tr>
<td>6. <strong>return</strong> Search(curr.right, target)</td>
<td>▶ recur on right subtree</td>
</tr>
<tr>
<td>7. <strong>else if</strong> target = curr.key <strong>then</strong></td>
<td></td>
</tr>
<tr>
<td>8. <strong>return</strong> curr</td>
<td>▶ successful search</td>
</tr>
</tbody>
</table>
Search: Iterative algorithm

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td><code>while curr ≠ null do</code></td>
</tr>
<tr>
<td>2.</td>
<td><code>if target &lt; curr.key then</code></td>
</tr>
<tr>
<td>3.</td>
<td><code>curr ← curr.left</code></td>
</tr>
<tr>
<td>4.</td>
<td><code>else if target &gt; curr.key then</code></td>
</tr>
<tr>
<td>5.</td>
<td><code>curr ← curr.right</code></td>
</tr>
<tr>
<td>6.</td>
<td><code>else if target = curr.key then</code></td>
</tr>
<tr>
<td>7.</td>
<td><code>return curr</code></td>
</tr>
<tr>
<td>8.</td>
<td><code>return null</code></td>
</tr>
</tbody>
</table>
Tree T:

Time per level

Total time: \( O(h) \)

Height

\( h \)

Runtime \( \in \Theta(h) \in O(n) \)
Insert 68
Insert 68
Insert: Recursive algorithm

\[ \text{INSERT}(\text{curr}, \text{item}) \]

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

1. if curr = null then
2. curr ← NODE(item)                  ▶ item does not exist

3. else if curr ≠ null then
4. if item < curr.key then
5. curr.left ← INSERT(curr.left, item) ▶ recur on left subtree
6. else if item > curr.key then
7. curr.right ← INSERT(curr.right, item) ▶ recur on right subtree
8. else if item = curr.key then
9. do nothing                             ▶ item exists

10. return curr
Insert: Iterative algorithm

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

1. `prev ← null`
2. **while** `curr ≠ null` **do**
3. `prev ← curr`
4. **if** `item < curr.key` **then**
   5. `curr ← curr.left`  ◀ recur on left subtree
5. `curr ← curr.left`
6. **else if** `item > curr.key` **then**
   7. `curr ← curr.right`  ◀ recur on right subtree
7. `curr ← curr.right`
8. **else if** `item = curr.key` **then**
9. **return** `curr`  ◀ item exists
10. `curr ← Node(item)`  ◀ item does not exist
11. **if** `prev ≠ null` **then**
12. **if** `item < prev.key` **then** `prev.left ← curr`
13. **if** `item > prev.key` **then** `prev.right ← curr`
14. **return** `curr`
Tree $T$:

- Height: $h$
- Time per level: $O(1)$
- Total time: $O(h)$

Runtime $\in \Theta(h) \in O(n)$
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
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

1. **if** \( curr = null \) **then**  
   - do nothing  
   - ▶ item does not exist
2. **else if** \( item < curr.key \) **then**  
   - \( curr.left \leftarrow \text{Delete}(curr.left, item) \)  
   - ▶ recur on left
3. **else if** \( item > curr.key \) **then**  
   - \( curr.right \leftarrow \text{Delete}(curr.right, item) \)  
   - ▶ recur on right
4. **else if** \( item = curr.key \) **then**  
   - ▶ item exists
5. **if** \( curr.left = null \) **then**  
   - \( curr \leftarrow curr.right \)  
   - ▶ 0 or 1 child
6. **else if** \( curr.right = null \) **then**  
   - \( curr \leftarrow curr.left \)  
   - ▶ 1 child
7. **else**  
   - \( curr.key \leftarrow \text{FindMax}(curr.left).key \)  
   - ▶ find predecessor
8. **else if** \( curr.right = null \) **then**  
   - ▶ 1 child
9. **else**  
   - \( curr.key \leftarrow \text{FindMax}(curr.left).key \)  
   - ▶ find predecessor
10. **else if** \( curr.right = null \) **then**  
   - ▶ 1 child
11. **else**  
   - \( curr.key \leftarrow \text{FindMax}(curr.left).key \)  
   - ▶ find predecessor
12. **return** curr
Delete: Iterative algorithm

Problem

How do you write an iterative algorithm for deleting an item?
Delete: Analysis

Tree $T$:

- Time per level: $O(1)$
- Total time: $O(h)$

Height $h$

Runtime $\in \Theta(h) \in O(n)$
Balanced Search Trees
## Balanced search trees: Motivation

<table>
<thead>
<tr>
<th>Data structure</th>
<th>Search</th>
<th>Insert</th>
<th>Delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary search tree</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Balanced search tree</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>
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.
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
Size and depth properties are satisfied.
Overflow: Size property is violated at [13 14 15 17].
Size property at [13 14 15 17] will be fixed via split operation.
Overflow: Size property is violated at [5 10 12 15].
Size property at [5 10 12 15] will be fixed via split operation.
Size and depth properties are satisfied.
Insert: Node split
Insert 4, 6, 12, 15
Insert 3, 5
Insert 10, 8
Underflow: Size property is violated is [4].
Size property will be fixed via transfer operation.
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].
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

<table>
<thead>
<tr>
<th>Method</th>
<th>Running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search</td>
<td>$\mathcal{O}(\log n)$</td>
</tr>
<tr>
<td>Insert</td>
<td>$\mathcal{O}(\log n)$</td>
</tr>
<tr>
<td>Delete</td>
<td>$\mathcal{O}(\log n)$</td>
</tr>
</tbody>
</table>
B Trees
Computer memory

- Network Storage
- External Memory
- Internal Memory
- Caches
- Registers
- CPU
## Cache-efficient algorithms: Example

<table>
<thead>
<tr>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>How do you efficiently sort a 1 GB file of natural numbers?</td>
</tr>
</tbody>
</table>
## Cache-efficient algorithms: Example

<table>
<thead>
<tr>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>How do you efficiently sort a 1 GB file of natural numbers?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Workout</th>
</tr>
</thead>
<tbody>
<tr>
<td>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 <strong>communication is more expensive than computation</strong>. 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: <strong>External-memory merge sort</strong>.</td>
</tr>
</tbody>
</table>
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., $T(n) \in \omega(S(n))$

• **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: $B = \text{data block size}$, $M = \text{cache size}$
## Cache-efficient algorithms

<table>
<thead>
<tr>
<th>Problem</th>
<th>Cache-inefficient algo</th>
<th>Cache-efficient algo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorting</td>
<td>Merge sort ( O(n \log n) )</td>
<td>Ext-memory merge sort ( O \left( \frac{n}{B} \log \frac{M}{B} \frac{n}{B} \right) )</td>
</tr>
<tr>
<td>Balanced tree</td>
<td>(2,4)-tree ( O(\log n) )</td>
<td>B tree ( O(\log_B n) )</td>
</tr>
<tr>
<td>Matrix product</td>
<td>Iterative ( O(n^3) )</td>
<td>Recursive D&amp;C ( O \left( \frac{n^3}{B\sqrt{M}} \right) )</td>
</tr>
</tbody>
</table>
\( (a, b) \)-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.
<table>
<thead>
<tr>
<th>Method</th>
<th>(2,4)-tree Communication</th>
<th>(2,4)-tree Computation</th>
<th>B tree Communication</th>
<th>B tree Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log_B n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Insert</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log_B n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Delete</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log_B n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

- 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