Data Structures – Trees
Sorting Algorithms

Informatik I für D-MAVT (FS 2013)
Christian Schumacher, chschuma@inf.ethz.ch

- Trees
- Binary Trees
- Binary Search Trees
- Sorting Algorithms
Linked Lists & Trees

- A linked list is a "tree" with degree 1
- A binary tree is a tree with degree 2
Definitions for Trees

- **root**
- **parent**
- **children**
- **node with degree 3**
- **leaves**
- Height or depth of the tree = 3
Tree Node Element Struct

- Like a list element but with more than one child (pointer)
- Value can be anything
- „Empty Tree“ is just a NULL pointer
- Example for binary tree node

```c
struct Node
{
    int value; // The value of the element
    Node *left; // Pointer to the left child
    Node *right; // Pointer to the right child
};

Node *root = NULL; // Pointer to root element
```
Use Recursive Functions

- A tree is a fractal:
  - If root is a (non-empty) tree, root->left and root->right will also be a tree
- A function that can handle the root node can also handle the child nodes
Use Recursive Functions

- Create a function that takes any tree element as a parameter
- Let the function print the data of the element
- Print the data of its child elements recursively

```c
void print_recursive( Node *node )
{
    cout << ... // print value

    // to print the childrens content
    print_recursive( node->left );
    print_recursive( node->right );
}
```
Use Recursive Functions

- Don't forget the base
- Otherwise: Endless loop or (here) NULL pointer dereferencing

```c
void print_recursive( Node *node )
{
    if( node != NULL )
    {
        cout << ...  
        print_recursive( node->left );
        print_recursive( node->right );
    }
}
```
Traversing Recursively

```c
void recursive( Node *node )
{
    if( node != NULL )
    {
        recursive( node->left ); //do something with this node
        recursive( node->right );
    }
}
```
Searching an Unsorted Tree

- Check if tree is empty
- Check if value of current node is the value we search
  - If so: return true;
- Else look in child nodes recursively
- Stop looking in child nodes when result was found or there are no more children
bool rec_search( Node *node, int i )
{
    if (node == NULL)
    {
        return false;
    }
    if (node->value == i)
    {
        return true;
    }
    return rec_search(node->left, i) || rec_search(node->right, i);
}
Deleting Tree Elements

- It's simple to create a memory leak

```c
void delete_element( Node * node )
{
    delete node;
}
```
Deleting Tree Elements

- Check for empty trees
- Recursively delete all child elements first
- After that delete current node

```c
void delete_element( Node *node )
{
    if( node != NULL )
    {
        delete_element( node->left );
        delete_element( node->right );
        delete node;
    }
}
```
Binary Search Tree

- Just like regular trees, but on the left there is always the smaller key and on the right the larger (or vice versa)
- Makes it much easier to search for a given key
Binary Search Tree Example

Binary search without recursion

```c
Node *next = root;
int search = 6;
while( next != NULL && next->value != search )
{
    if( next->value < search )
    {
        next = next->right;
    }
    else
    {
        next = next->left;
    }
}
```
Inserting element into tree

- Create new node
  
  ```
  Node *node = new Node;
  node->value = 42;
  node->right = NULL;
  node->left = NULL;
  ```

- Inserting into search tree:
  - Search for value you want to insert until reaching null
  - Change the address of the null pointer to the address of your new node
Sorting

- Frequently used in software
  - File Manager: Detail View
  - Task Manager: Sort by RAM usage
  - Managing large music collections
- Task: Sort a sequence of \( n \) data elements
- The larger \( n \), the more time \( t \) it takes to sort the sequence
# Efficiency

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Selection</th>
<th>Bubble</th>
<th>Merge</th>
<th>Quick</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst case</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Best case</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Average case</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
</tbody>
</table>
Selectionsort

1. Find the **minimum value** in the list
2. **Swap it** with the value in the **first position**
3. **Repeat** the steps above for the remainder of the list (starting at the second position and advancing each time)
Selectionsort
Selectionsort

```c
void selectionSort(int *array, int length) // selection sort function
{
    int i, j, min, minat;
    for (i = 0; i < (length - 1); i++)
    {
        minat = i;
        min = array[i];
        for (j = i + 1; j < (length); j++) // select the min of the rest of array
        {
            if (min > array[j]) // ascending order for descending reverse
            {
                minat = j; // the position of the min element
                min = array[j];
            }
        }
        int temp = array[i];
        array[i] = array[minat]; // swap
        array[minat] = temp;
    }
}
```
Selectionsort
Bubblesort

- Simple sorting algorithm
  - Repeatedly step through the sequence
  - Compare each pair of adjacent items and swap them if they are in the wrong order
- The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.
Bubblesort

6 5 3 1 8 7 2 4
Bubblesort
Mergesort

- Divide and conquer algorithm
- Divide the unsorted list into \textit{n sublists}, each containing 1 element (a list of 1 element is considered sorted).
- \textbf{Repeatedly merge} sublists to produce new sublists until there is only 1 sublist remaining.
Mergesort

38 27 43 3 9 82 10

38 27 43 3

9 82 10

38 27

43 3

9 82

10

27 38

43 3

9 82

10

3 43

9 82

10

3 27 38 43

9 10 82

3 9 10 27 38 43 82
Mergesort

6 5 3 1 8 7 2 4
Quicksort

- Divide and conquer algorithm.
  - **Divide** a large list into **two smaller sub-lists**: the **low** elements and the **high** elements.
  - **Recursively** sort the sub-lists

- The steps are:
  - Pick an element, called a **pivot**, from the list.
  - Reorder the list so that all elements with values **less than the pivot come before** the pivot, while all elements with values **greater than the pivot come after** it
  - After this partitioning, the pivot is in its final position. This is called the **partition operation**.
  - **Recursively** sort the sub-list of lesser elements and the sub-list of greater elements.
Quicksort

6 5 3 1 8 7 2 4
Quicksort
Übung 12

- Aufgabe 1: Binärer Suchbaum
  - void insert ( Node * node , int value )
    - Falls value den gleichen Wert wie der aktuelle Knoten hat, wurde der Wert schon eingefügt → Funktion beenden
    - Falls value kleiner ist als der Wert des Knotens, überprüfe, ob der Knoten einen linken Teilbaum hat
      - Falls ja: Rekursiver Funktionsaufruf: insert( node->left , value );
      - Falls nein: Erstelle einen neuen Knoten, setze ihn als linken Teilbaum
    - Falls value grösser ist als der Wert des Knotens, überprüfe, ob der Knoten einen rechten Teilbaum hat
      - Falls ja: Rekursiver Funktionsaufruf: insert( node->right, value );
      - Falls nein: Erstelle einen neuen Knoten, setze ihn als rechten Teilbaum
Übung 12

- Aufgabe 1: Binärer Suchbaum
  - Daten einlesen

```cpp
#include <iostream>
#include <fstream>

int main() {
    std::ifstream file;
    file.open("input.txt");
    while ( !file.eof() ) {
        int a;
        file >> a;
        // processing...
    }
    file.close();
    return 0;
}
```