Numerical Simulation and Scientific Computing I

Lecture 9: Algorithm Analysis and Data Structures

Clemens Etl, Paul Manstetten, and Josef Weinbub

Institute for Microelectronics
TU Wien

nssc@iue.tuwien.ac.at

WS 2023

Quiz

- Q1: Consider a 32-bit LCG. How many random numbers can you generate before the sequence repeats?

32-bit → m= 2^{32}:

Periodicity = 2^{32} → $\sim 10^9$ random numbers

Quiz

- Q2: What are the consequences of entropy pool depletion? How would you handle it?

The entropy pool is unable to provide more random bits. I could wait until it is replenished.

Quiz

- Q3: What are the requirements to initialize correctly one independent RNG per thread?

Each thread must initialize its own RNG with a different seed

Quiz

- Q4: Which STL containers are usually implemented using hashes? What is the advantage of doing so?
  Later.
- Q5: How are elements accessed on a std::list? (What is the consequence of this for an OpenMP parallel for?)
  Later.

Outline

- Algorithm Analysis
  - “Big O” Notation
- Data Structures
  - Arrays
  - Lists
  - Trees
  - Hashes
  - C++ STL

Main References

- Basic Concepts in Data Structures
  - Author: Shmuel T. Klein
  - https://catalogplus.tuwien.at/permalink/f/8agg25/TN_cdi_askewsholts_vlebooks_9781316883716
  - eBook available: https://www.cambridge.org/core/books/basic-concepts-in-data-structures/658E935CC9790488B4B4BF797EFC2101

- C++ STL Containers library
  - https://en.cppreference.com/w/cpp/container

Additional References

- C++ plus Data Structures
  - Authors: N. Dale, C. Weems, T. Richards
  - eBook available: https://catalogplus.tuwien.at/permalink/f/8agg25/TN_cdi_proquest_ebookcentral_EBC4714314
- Data Structures & Algorithms in C++
  - Authors: M. T. Goodrich, R. Tamassia, D. Mount
  - eBook available: https://catalogplus.tuwien.at/permalink/f/8agg25/TN_cdi_safari_books_9780470383278

Take-home Message

- “Big O” analysis is a useful tool for comparing algorithms
  - Scale: $O(1)$ $O(\log n)$ $O(n)$ $O(n \log n)$ $O(n^2)$ $O(2^n)$ $O(n!)$
- Unless your problems require ordering, hashes are usually more efficient
  - Calculating is usually better than comparing
- Always consider how your containers are distributed in memory

Algorithms & Data Structures

- Definitions [Goodrich et al.]:

- **Algorithms** are step-by step procedure for performing some task in a finite amount of time

- **Data Structures** are a systematic way of organizing and accessing data

Algorithm Analysis – "Big O"

- Experimental studies are not always straightforward
- Asymptotic analysis of the (pseudo-)code: How much does it grow with problem size?
- Counting how many primitive operations as a function of the input size n
- Definition: $f(n)$ is $O(g(n))$ if for $c > 0$

$$f(n) \leq cg(n), \text{ for } n > n_0$$

Primitive Operations [Goodrich et al.]:

- Assigning a value to a variable
- Calling a function
- Performing an arithmetic operation
- Comparing two numbers
- Indexing into an array
- Following an object reference
- Returning from a function

“Big O” – Example 1

```
arrayMax(A,n) {
        currentMax = A[0]
        for i in [1,n)
                if A[i] > currentMax
                        currentMax = A[i]
        return currentMax
}
```

- Total number of primitive operations: $8n - 2$ -> $O(n)$
- “Scale”: $O(1)$ $O(\log n)$ $O(n)$ $O(n \log n)$ $O(n^2)$ $O(2^n)$
- Things to watch out:
  - O is the same notation in FD…
  - Depending on the constant, sometimes the “scale” doesn’t hold up
  - Also: not all “primitive operations” are created equal

Source: Goodrich et al.: Data Structures & Algorithms in C++, 2nd ed., 2011, Wiley

"Big O" – Example 2 – Poll 9

```
prefixAverages(X) {
        //Output, A[i] is the average of
        // X[0],...,X[i]
        for i in [1,n)
                a = 0
                for j in [0,i)
                        a += X[j]
                A[i] = a/(i + 1)
        return A
}
```

- A) $O(\log n)$
- B) $O(n)$
- C) $O(n \log n)$
- D) $O(n^2)$
- E) $O(n!)$

“Big O” – Example 2 – Poll 9

```
prefixAverages(X) {
        //Output, A[i] is the average of
        // X[0],...,X[i]
        for i in [1,n)
                a = 0
                for j in [0,i)
                        a += X[j]
                A[i] = a/(i + 1)
        return A
}
```

- A) $O(\log n)$
- B) $O(n)$
- C) $O(n \log n)$
- **D) $\boldsymbol{O(n^2)}$**
- E) $O(n!)$

Data Structures

- Arrays
- Lists
- Trees
- Hashes
- C++ STL

Arrays

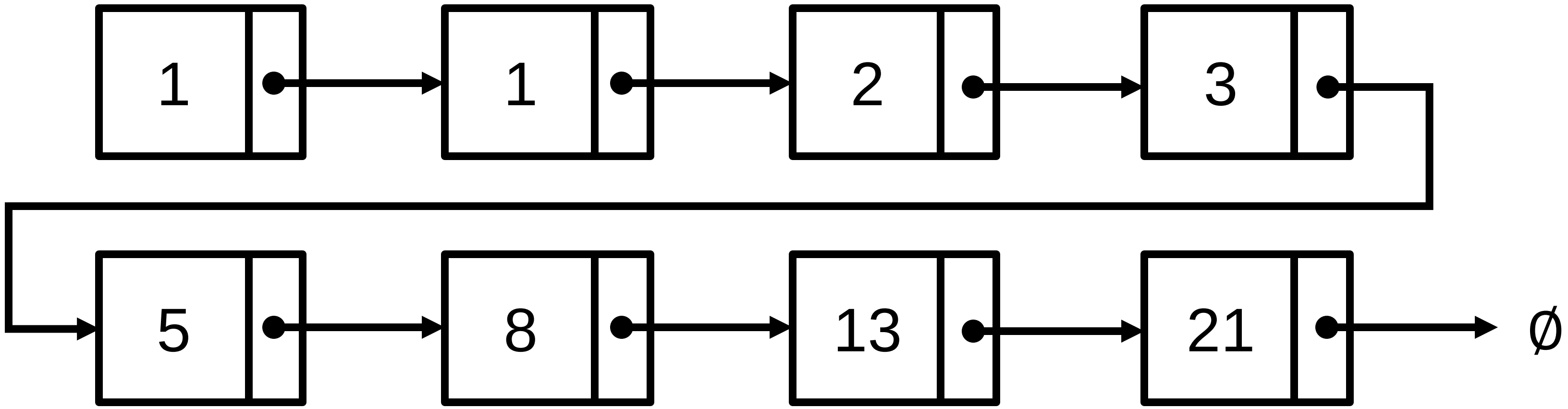
`T`	1	1	2	3	5	8	13	21
	0	1	2	3	4	5	6	7

- A collection of elements of the same type
- Usually:
  - Accessible by square brackets and an integer index: `T[6]==13`
  - Contiguous in memory
- Simple random access
  - However: inserting can be challenging

Quiz – Q4

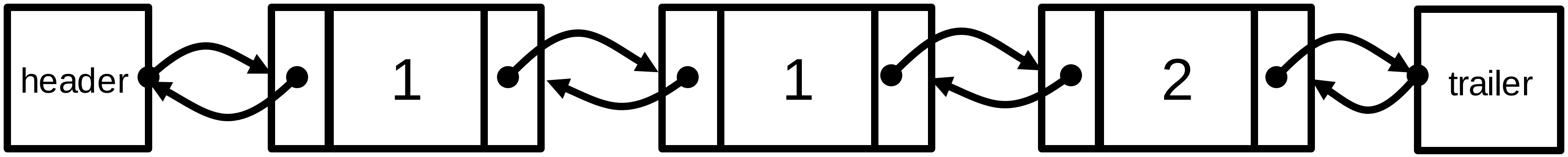
- **How are elements accessed on a `std::list`?**

Lists



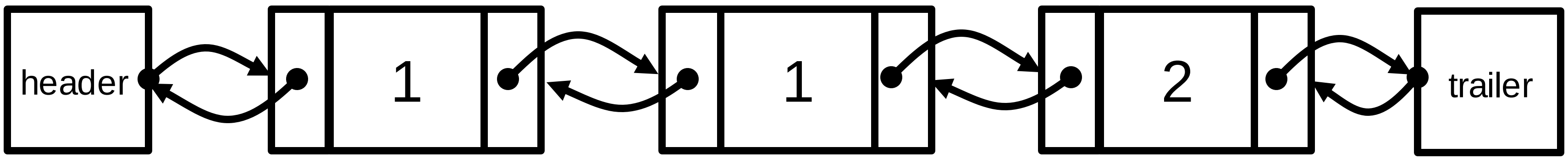
- A (singly-linked) list is a collection of nodes, each storing its element and a pointer to the next node
- Advantage: very simple to insert
- Disadvantage: scattered in memory

Doubly-Linked Lists



- What is the advantage vs. singly linked list?

Doubly-Linked Lists



- What is the advantage vs. singly linked list?
  - Iterate forward and backward

Quiz – Q4

- How are elements accessed on a `std::list`? **What is the consequence of this for an OpenMP parallel for?**

Quiz – Q4 – Poll 10

- How are elements accessed on a `std::list`? **Is it possible to use OpenMP parallel for to iterate it?**
  - A) Yes
  - B) No

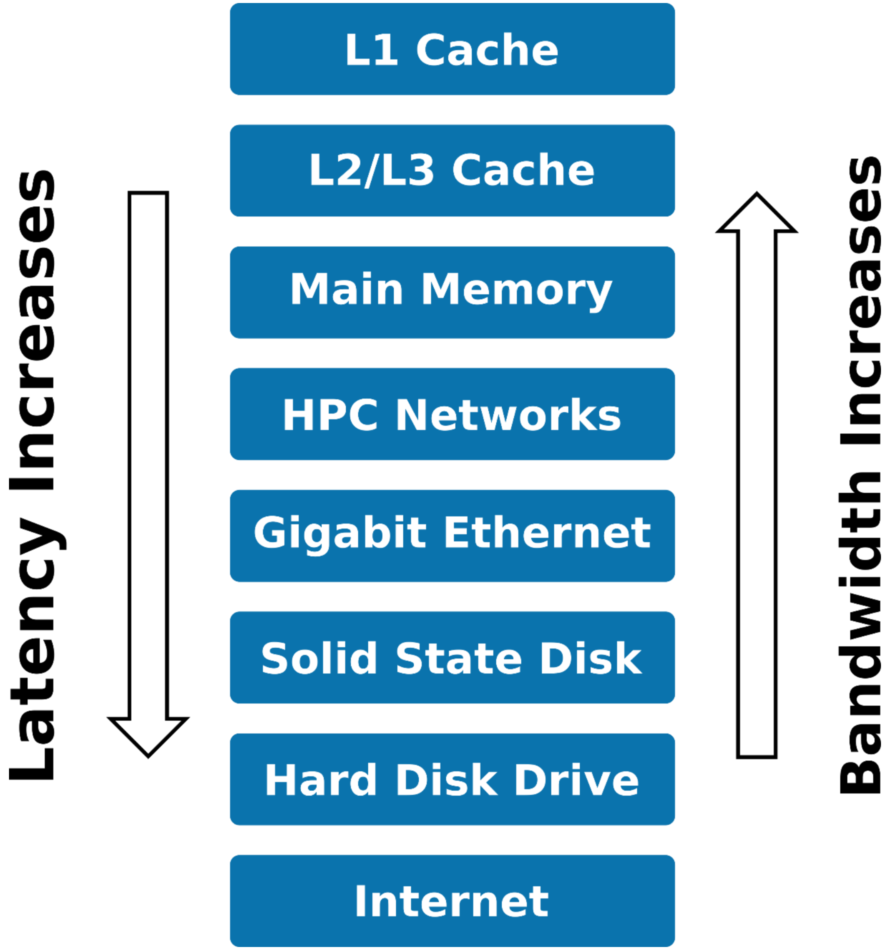
Quiz – Q4 – Poll 10

- How are elements accessed on a `std::list`? **Is it possible to use OpenMP parallel for to iterate it?**
  - A) Yes
  - **B) No**

Sequence Containers in the STL

- `std::array`: static contiguous array
  - Random access: $O(1)$, Insertion: N/A
- `std::vector`: dynamic contiguous array
  - Random access: $O(1)$, Insertion: $O(n)$, but usually $O(1)$ at the end
- `std::forward_list`: singly-linked list
  - Random access: N/A, Insertion: $O(1)$
- `std::list`: doubly-linked list
  - Random access: N/A, Insertion: $O(1)$
- `std::deque`: double-ended queue
  - Random access: $O(1)$, Insertion: $O(1)$ at ends and $O(n)$ elsewhere

Sidenote: Memory Hierarchy

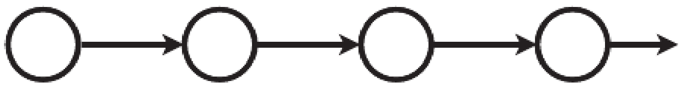


- We always want to maximize temporal and spatial locality
- “Efficiency with Algorithms, Performance with Data Structures”
  - https://youtu.be/fHNmRkzxHWs

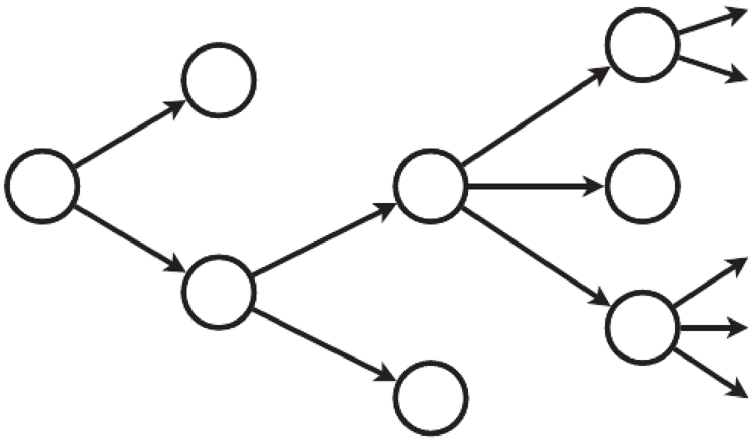
STL Iterators

- Iterators abstract the process of scanning a container
- The container offers: `begin()` and `end()`
- An iterator `it` is “like a pointer”: `*it` and `++it`
- Some types of iterators in C++
  - Input: `*it` and `++it` (single pass)
  - Forward: `++it` (multiple passes)
  - Bidirectional: `--it`
  - Random Access: supports `it+i` and `it-i`

Trees



List

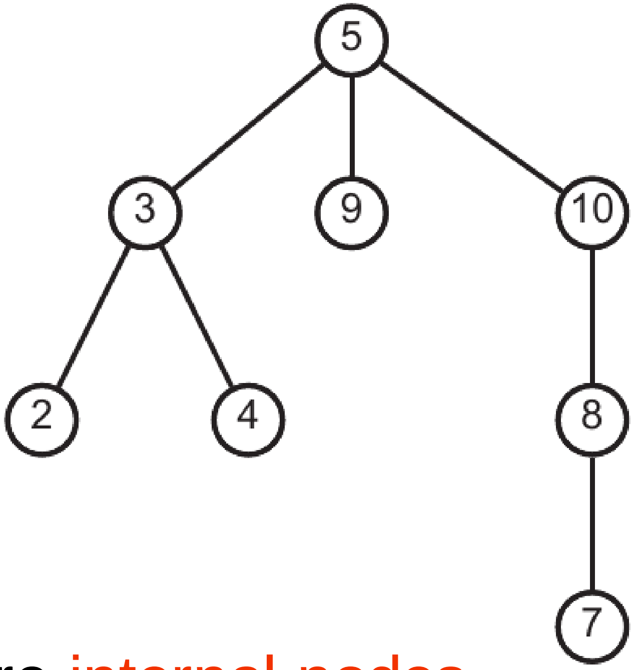


Tree

Source: Klein: Basic Concepts in Data Structures, 1st ed., 2016, CUP

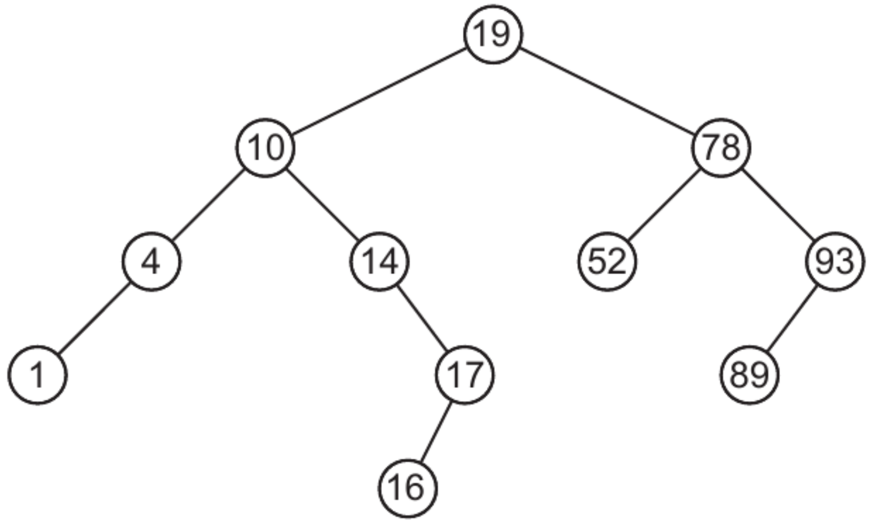
Nomenclature

- 5 is the root
- 3 is the parent of 4
  - 4 is a child of 3
- 2 and 4 are siblings
- 2, 4 and 7 are leaves, the remaining are internal nodes



Source: Klein: Basic Concepts in Data Structures, 1st ed., 2016, CUP

Binary Search Trees (BST)



- Binary: each node has at most a left child and a right child

- Search: for a given node, values on the left subtree are smaller and, on the right, larger

Source: Klein: Basic Concepts in Data Structures, 1st ed., 2016, CUP

Key-Value Pairs

- Given a complicated value, assign a key (sometimes unique)

- For example:
  - Information about MSc students (name, email, grades…): value
  - Unique ID (Matrikelnr.): key

- Keys must be sortable
  - This means we can put any type of data on a BST!

Associative Containers in the STL

- All of them are usually implemented as BSTs (commonly red-black trees)
  - They all are $O(\log n)$ for search, insertion and removal
- `std::map`: key-value pairs, sorted by unique keys
  - Has the convenient `[]` operator!
  - How is an `std::map` in memory?
- `std::set`: sorted unique keys
- `std::multimap`: key-value pairs, sorted by keys
- `std::multiset`: sorted keys

Quiz – Q5

- Which STL containers are usually implemented using hashes? What is the advantage of doing so?

Problem

- We don't always need sorted keys
  - Additional computational cost!

Hashes

- Insight: calculate instead of compare
- Hash function $h(X)$ maps from all possible keys to $[0, M)$
  - Where M is the number of entries
- The resulting table is called a hash table
- Collisions $(h(X) = h(Y))$ are unavoidable -> rehashing
- What do we gain and lose wrt. sorted containers?

Hashes

- Insight: calculate instead of compare
- Hash function $h(X)$ maps from all possible keys to $[0, M)$
  - Where M is the number of entries
- The resulting table is called a hash table
- Collisions $(h(X) = h(Y))$ are unavoidable -> rehashing
- What do we gain and lose wrt. sorted containers?
  - Lose: order, worst-case performance
  - Win: (some) memory locality, average performance

Unordered Associative Containers in the STL

- On average, they are all $O(1)$ for search, insertion and removal.
  - But can be $O(n)$ on the worst case. Why?
- `std::unordered_map`: key-value pairs, hashed by unique keys
- `std::unorderd_set`: hashed, unique keys
- `std::unordered_multimap`: key-value pairs, hashed by keys
- `std::unordered_multiset`: hashed keys

Recap – Contiguous in memory – Poll 11

- Which set have ALL elements contiguous in memory?
- A) `std::array, std::map, std::unordered_map`
- B) `std::list, std::map, std::set`
- C) `std::vector, std:unordered_map, std:array`
- D) `std::array, std::unordered_set, std::list`

Recap – Contiguous in memory – Poll 11

- Which set have ALL elements contiguous in memory?
- A) `std::array, std::map, std::unordered_map`
- B) `std::list, std::map, std::set`
- **C) `std::vector, std:unordered_map, std:array`**
- D) `std::array, std::unordered_set, std::list`

Take-home Message – Version 2

- "Big O" analysis is a useful tool for comparing algorithms
  - Scale: $O(1)$ $O(\log n)$ $O(n)$ $O(n \log n)$ $O(n^2)$ $O(2^n)$ $O(n!)$
  - But remember that the constants still matter…
- Unless your problems require ordering, hashes are usually more efficient
  - Calculating is usually better than comparing
- Always consider how your containers are distributed in memory
  - Performance comes from exploiting temporal and spatial locality
- `std::vector` is your friend

Quiz

- Q1: What is the “Big O” for the symmetric CCS-vector product from Exercise 3?

- Q2: What is the difference between inserting an element in the middle of an array and in the middle of a linked list?

- Q3: What are disadvantages of hash tables compared to BSTs?

- Q4: What is the difference between a process and a thread? Which resources can be shared among them?

- Q5: What does the directive `#pragma omp atomic` do?

Next stop

- Shared memory parallel optimization
- OpenMP