Structure

Sorted Sequences

Linked Lists

Binary Search Trees
Structure:
■ We have a set of keys mapped to values
■ We have an ordering $<$ applied to the keys
■ We need the following operations:
  ■ \texttt{insert(key, value)}: insert the given pair
  ■ \texttt{remove(key)}: remove the pair with the given key
  ■ \texttt{lookup(key)}: find the element with the given key, if it is not available find the element with the next smallest key
  ■ \texttt{next()}/\texttt{previous()}: returns the element with the next bigger/smaller key. This enables iteration over all elements
**Sorted Sequences**

**Introduction**

**Application examples:**

- Example: database for books, products or apartments
- Large number of records (data sets / tuples)
- Typical query: return all apartments with a monthly rent between 400€ and 600€
  - This is called a **range query**
  - We can implement this with a combination of `lookup(key)` and `next()`
  - It’s not essential that an apartment exists with **exactly** 400€ monthly rent
- We do not want to sort all elements every time on an **insert** operation
- How could we implement this?
Sorted Sequences
Implementation 1 (not good) - Static Array

Static array:

```
3 5 9 14 18 21 26 40 41 42 43 46
```

- **lookup** in time $O(\log n)$
  - With **binary search**
  - Example: `lookup(41)`

- **next / previous** in time $O(1)$
  - They are next to each other

- **insert** and **remove** up to $\Theta(n)$
  - We have to copy up to $n$ elements
Hash map:

- **insert** and **remove** in $O(1)$
  
  If the hash table is big enough and we use a good hash function

- **lookup** in time $O(1)$
  
  If element with **exactly** this key exists, otherwise we get **None** as result

- **next** / **previous** in time up to $\Theta(n)$
  
  Order of the elements is independent of the order of the keys
Sorted Sequences
Implementation 3 (good?) - Linked List

Linked list:

- Runtimes for doubly linked lists:
  - next / previous in time $O(1)$
  - insert and remove in $O(1)$
  - lookup in time $\Theta(n)$

- Not yet what we want, but structure is related to binary search trees
- Let’s have a closer look
Linked list:
- Dynamic datastructure
- Number of elements changeable
- Data elements can be simple types or composed data structures
- Elements are linked through references / pointer to the predecessor / successor
- Single / doubly linked lists possible

**Figure**: Linked list
Properties in comparison to an array:

- Minimal extra space for storing pointer
- We do not need to copy elements on insert or remove
- The number of elements can be simply modified
- No direct access of elements
  ⇒ We have to iterate over the list
List with head / last element pointer:

- Head element has pointer to first list element
- May also hold additional information:
  - Number of elements

**Figure:** Singly linked list
Doubly linked list:

- Pointer to successor element
- Pointer to predecessor element
- Iterate forward and backward

Figure: Doubly linked list
class Node:
    """ Defines a node of a singly linked list. """

    def __init__(self, value, nextNode=None):
        self.value = value
        self.nextNode = nextNode
Creating linked lists - Python:

- `first = Node(7)`
  
  ![Diagram of a single node with value 7]

- `first.nextNode = Node(3)`
  
  ![Diagram of two nodes with values 7 and 3]

- `first.nextNode.value = 4`
  
  ![Diagram of two nodes with values 7 and 4]
Inserting a node after node \texttt{cur}: 

```
first -> n_0 -> n_1 -> n_2 -> n_3 -> None
```

\textbf{cur}
Inserting a node after node \texttt{cur}:

\begin{itemize}
\item \texttt{ins} = \texttt{Node(n)}
\end{itemize}
Inserting a node after node \( \text{cur} \):

\[
\text{ins.nextNode} = \text{cur.nextNode}
\]
Inserting a node after node `cur`:  

- `cur.nextNode = ins`
Inserting a node after node \texttt{cur} - single line of code:

\texttt{cur.nextNode = Node(value, cur.nextNode)}
Removing a node \texttt{cur}: 

\[
\text{first} \quad n_0 \quad n_1 \quad n_2 \quad n_3 \quad \text{None}
\]
Removing a node \texttt{cur:}

- Find the predecessor of \texttt{cur}:

  \[
  \texttt{pre = first} \\
  \texttt{while pre.nextNode != cur:} \\
  \hspace{1cm} \texttt{pre = pre.nextNode}
  \]

- Runtime of $O(n)$
- Does not work for first node!
Removing a node `cur`:

- Update the pointer to the next element:
  ```python
  pre.nextNode = cur.nextNode
  ```
- `cur` will get destroyed automatically if no more references exist (`cur=None`)

```python
first -> n0 -> n1 -> n2 -> n3 -> None
```
Removing the first node:

- Update the pointer to the next element:
  `first = first.nextNode`
- `cur` will get automatically destroyed if no more references exist (`cur=None`)
Removing a node `cur`: (General case)

```python
if cur == first:
    first = first.nextNode
else:
    pre = first
    while pre.nextNode != cur:
        pre = pre.nextNode
    pre.nextNode = cur.nextNode
```
Using a head node:

- **Advantage:**
  - Deleting the first node is no special case

- **Disadvantage:**
  - We have to consider the first node at other operations
  - Iterating all nodes
  - Counting of all nodes
  - …
class LinkedList:
    def __init__(self):
        self.itemCount = 0
        self.head = Node()
        self.last = self.head

    def size(self):
        return self.itemCount

    def isEmpty(self):
        return self.itemCount == 0
def append(self, value):
    ...

def insertAfter(self, cur, value):
    ...

def remove(self, cur):
    ...

def get(self, position):
    ...

def contains(self, value):
    ...
Head, last:

- Head points to the first node, last to the last node.
- We can append elements to the end of the list in $O(1)$ through the last node.
- We have to keep the pointer to last updated after all operations.
Appending an element:

```python
def append(self, value):
    last.nextNode = Node(value)
    last = last.NextNode
    itemCount += 1
```

- The pointer to `last` avoids the iteration of the whole list
Inserting after node `cur`:

![Diagram showing the insert after operation on a linked list with nodes labeled 0, 1, ..., n. The `ins` node is inserted after the current node `cur`. The `value` field of the `ins` node is set to None.](image)
Inserting after node \texttt{cur}:

- The pointer to \texttt{head} is not modified

```python
def insertAfter(self, cur, value):
    if cur == last:
        # also update last node
        append(value)
    else:
        # last node is not modified
        cur.nextNode = Node(value, \  
cur.nextNode)
    itemCount += 1
```
Remove node \textit{cur}:
**Remove node** `cur`:

- Searching the predecessor in $O(n)$

```python
def remove(self, cur):
    pre = first
    while pre.nextNode != cur:
        pre = pre.nextNode

    pre.nextNode = cur.nextNode
    itemCount -= 1

    if pre.nextNode == None:
        last = pre
```
Getting a reference to node at pos:

- Iterate the entries of the list until position in $O(n)$

```python
def get(self, pos):
    if pos < 0 or pos >= itemCount:
        return None
    cur = head
    for i in range(0, pos):
        cur = cur.nextNode
    return cur
```
Searching a value:

- First element is head without an assigned value
- Iterate the entries of the list until value found in $O(n)$

```python
def contains(self, value):
    cur = head

    for i in range(0, itemCount):
        cur = cur.nextNode
        if cur.value == value:
            return True

    return False
```

Linked Lists

Runtime

Runtime:
- Singly linked list:
  - `next` in $O(1)$
  - `previous` in $\Theta(n)$
  - `insert` in $O(1)$
  - `remove` in $\Theta(n)$
  - `lookup` in $\Theta(n)$
- Better with **doubly linked lists**
Doubly linked list:

- Each node has a reference to its successor and its predecessor
- We can iterate the list forward and backward
Doubly linked list:

- It is helpful to have a head node.
- We only need one head node if we cyclically connect the list.

![Doubly Linked List Diagram]

```
head
```

```
0          1          ...          n
```
Runtime of doubly linked list:

- **next** and **previous** in $O(1)$
  - Each element has a pointer to predecessor/successor

- **insert** and **remove** in $O(1)$
  - A constant number of pointers needs to be modified

- **lookup** in $\Theta(n)$
  - Even if the elements are sorted we can only retrieve them in $\Theta(n)$
    - Why?
Linked list in book:

head

0 → 1 → 2 → 3
Linked list in memory:

```
0x06970641 0x01D5A0BC 0x192D8203
3

head

0x01D5A0BC 0x1695FE08 0x06970641
0x01D5A0BC

0x1695FE08 0x01637E26
0

0x192D8203 0x06970641 0x1637E26
2

0x1695FE08 0x01637E26
1
```
Runtime of a search tree:

- **next** and **previous** in $O(1)$
  Pointers corresponding to linked list

- **insert** and **remove** in $O(\log n)$

- **lookup** in $O(\log n)$
  The structure helps searching efficiently
Idea:

- We define a total order for the search tree
- All nodes of the left subtree have smaller keys than the current node
- All nodes of the right subtree have bigger keys than the current node
Edge direction indicates ordering

Figure: a binary search tree
Binary Search Trees

Introduction

Figure: another binary search tree
Binary Search Trees

Introduction

Figure: **not** a binary search tree
**Implementation:**

- For the heap we had all elements stored in an array.
- Here we link all nodes through pointers/references, like linked lists.
- Each node has a pointer/reference to its children (leftChild/rightChild).
- None for missing children.
Implementation:
- We create a sorted doubly linked list of all elements
- This enables an efficient implementation of (next / previous)
Lookup:

- Definition:
  “Search the element with the given key. If no element is found return the element with the next (bigger) key.”

- We search from the root downwards:
  - Compare the searched key with the key of the node
  - Go to the left / right until the child is `None` or the key is found
  - If the key is not found return the next bigger one
For each node applies the total order:
keys of left subtree < node.key < keys of right subtree

Examples:
lookup(14)
lookup(6)
lookup(19)

Figure: binary search tree with total order “<”
Insert:

- We search for the key in our search tree
- If a node is found we replace the value with the new one
- Else we insert a new node
- If the key was not present we get a None entry
- We insert the node there

Figure: Binary search tree with total order "<"
**Remove:** case 1: the node “5” has no children

- Find **parent** of node “5” (“6”)
- Set left / right child of node “6” to **None** depending on position of node “5”

![Binary search tree diagram](image_url)

**Figure:** Binary search tree with total order “<”
**Remove:** Case 1: The node “5” has no children

- Find **parent** of node “5” ("6")
- Set left / right child of node “6” to **None** depending on position of node “5”

**Figure:** binary search tree after deleting node “5”
**Remove:** Case 2: The node “12” has one child

- Find the **child** of node “12” (“14”)
- Find the **parent** of node “12” (“8”)
- Set left / right **child** of node “8” to “14” depending on position of node “12” (skip node “14”)

**Figure:** binary search tree with total order “<”
Remove: Case 2: The node “12” has one child

- Find the child of node “12” (“14”)
- Find the parent of node “12” (“8”)
- Set left / right child of node “8” to “14” depending on position of node “12” (skip node “14”)

Figure: binary search tree after deleting node “12”
Remove: Case 3: The node “4” has two children

- Find the **successor** of node “4” (“5”)
- Replace the value of node “4” with the value of node “5”
- Delete node “5” (the **successor** of node “4”) with remove-case 1 or 2
- There is no left node because we are deleting the predecessor
Remove: Case 3: The node “4” has two children

- Find the successor of node “4” (“5”)
- Replace the value of node “4” with the value of node “5”
- Delete node “5” (the successor of node “4”) with remove-case 1 or 2
- There is no left node because we are deleting the predecessor
How long takes **insert** and **lookup**?

- Up to $\Theta(d)$, with $d$ being the depth of the tree (The longest path from the root to a leaf)
- **Best case** with $d = \log n$ the runtime is $\Theta(\log n)$
- **Worst case** with $d = n$ the runtime is $\Theta(n)$
- If we **always** want to have a runtime of $\Theta(\log n)$ then we have to **rebalance** the tree

![Figure: degenerated binary tree $d = n$](image1)

![Figure: complete binary tree $d = \log n$](image2)
Course literature


- **Linked List**
  
  [Wik] Linked list
  
  https://en.wikipedia.org/wiki/Linked_list

- **Binary Search Tree**
  
  [Wik] Binary search tree
  
  https://en.wikipedia.org/wiki/Binary_search_tree