Algorithms & Data Structures Notes - SoSe 24

Igor Dimitrov

2024-04-22

Table of contents

Preface

This is a Quarto book.

To learn more about Quarto books visit https://quarto.org/docs/books.

Part I

Introduction

1 Program Run-time Analysis

1.1 Reccurence Relations

Consider a very simple reccurence relation:

$$T(n) := \begin{cases} 1 & n = 1 \\ n + T(n-1), & n > 1 \end{cases}$$

With **mathematical induction** we can formally show that $T(n)$ is quadratic. But there is a simpler & more intuitive way:

$$\begin{aligned} T(n) &= n + T(n-1) && \text{(Def } T(\cdot)\text{)} \\ &= n + n - 1 + T(n-2) \\ &= ... && \text{(Repeat } n-2 \text{ times)} \\ &= n + n - 1 + \cdots + T(1) \\ &= n + n - 1 + \cdots + 1 && \text{(Def } T(1)\text{)} \\ &= \frac{n(n+1)}{2} && \text{(Gauss)} \\ &\in \mathcal{O}(n^2) \end{aligned}$$

This method can be applied to the more complex divide-and-conquer reccurence relation from the lecture:

$$R(n) := \begin{cases} a, & n = 1 \\ c\dot{n} + d \cdot R(\frac{n}{b}), & n > 1 \end{cases}$$

Applying the above method we expand $R(\cdot)$ repetitively according to its definition until we reach the base case, rearranging terms when necessary:

$$
\begin{aligned}
R(n) &= c \cdot n + d \cdot R(\frac{n}{b}) && \text{(Def } R(\cdot)) \\
&= c \cdot n + d(c\frac{n}{b} + d \cdot R(\frac{n}{b^2})) \\
&= c \cdot n + d\Big(c\frac{n}{b} + d \cdot (c \cdot \frac{n}{b^2} + d \cdot R(\frac{n}{b^2}))\Big) \\
&= c \cdot n + d \cdot c\frac{n}{b} + d^2 c\frac{n}{b^2} + d^3 \cdot R(\frac{n}{b^3}) && \text{(Rearrange)} \\
&= c \cdot n \left(1 + \frac{d}{b} + \frac{d^2}{b^2}\right) + d^3 \cdot R(\frac{n}{b^3}) \\
&= ... && \text{(Repeat } k\text{-times)} \\
&= c \cdot n \left(1 + \frac{d}{b} + \cdots + \frac{d^{k-1}}{b^{k-1}}\right) + d^k \cdot R(\frac{n}{b^k}) \\
&= c \cdot n \sum_{i=0}^{k-1} \left(\frac{d}{b}\right)^i + d^k \cdot R(\frac{n}{b^k}) \\
&= c \cdot n \sum_{i=0}^{k-1} \left(\frac{d}{b}\right)^i + d^k \cdot R(1) && \text{(Ass } \tfrac{n}{b^k} = 1) \\
&= c \cdot n \sum_{i=0}^{k-1} \left(\frac{d}{b}\right)^i + a \cdot d^k && \text{(Def } R(1))
\end{aligned}
$$

See lecture slides for the complexity analysis of final expression.

1.2 Master Theorem

For reccurence relations of the form:

$$
T(n) := \begin{cases} a, & n = 1 \\ b \cdot n + c \cdot T(\frac{n}{d}), & n > 1 \end{cases}
$$

Master theorem gives the solutions:

$$
T(n) = \begin{cases} \Theta(n), & c < d \\ \Theta(n \log(n)), & c = d \\ \Theta(n^{\log_b(d)}), & c > d \end{cases}
$$

Example: **Merge Sort**.

Complexity of merge sort satisfies the reccurence relation:

$$T(1) = 1$$
$$T(n) = \mathcal{O}(n) + 2 \cdot T(\frac{n}{2})$$

Thus with $c = 2 = d$ the second case of MT applies: $T(n) = \Theta(n \log n)$

1.3 Amortized Analysis

Part II

Data Structures

2 Lists

2.1 Sequences as Arrays and Lists

Many terms for same thing: sequence, field, list, stack, string, **file**… Yes, files are simply sequences of bytes!

three views on lists:

- **abstract**: $(2, 3, 5, 7)$
- **functionality**: stack, queue, etc… What operations does it support?
- **representation**: How is the list represented in a given programming model/language/paradigm?

2.2 Applications of Lists

- Storing and processing any kinds of data
- Concrete representation of abstract data types such as: set, graph, etc…

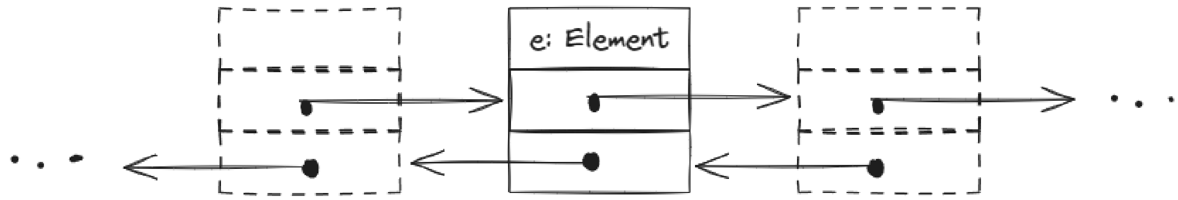
2.3 Linked and Doubly Linked Lists

	simply linked	doubly linked
lecture	`SList`	`List`
c++	`std::forward_list`	`std::list`

Doubly linked lists are usually **simpler** and require "only" double the space at most. Therefore their use is more widespread.

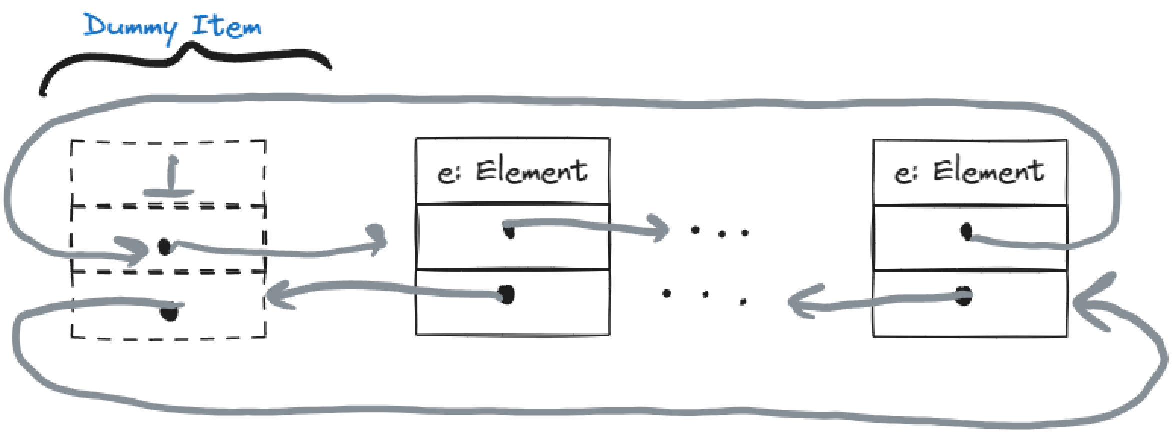
2.3.1 List Items

```
Class Item of T :=
    e: T //Data item of type T
    next: *Item //Pointer to Item
    prev: *Item //Pointer to Item
    invariant next->prev = this = prev->next
```



Problem: * predeccessor of first list element? * successor of last list element?

Solution: Dummy Item with an empty data field as follows:



Advatanges of this solution:

- **Invariant** is always satisfied
- Exceptions are avoided, thus making the coding more:
    - simple
    - readable
    - faster
    - elegant

Disadvantages: a little more storage space.

2.3.2 The List Class

```
Class List of T :=
    dummy := (
        Null : T
        &dummy : *T // initially list is empty, therefore next points to the
↪   dummy itself
        &dummy : *T // initially list is empty, therefore prev points to the
↪   dummy itself
    ) : Item

    // returns the address of the dummy, which represents the head of the
↪   list
    Function head() : *Item :=
        return address of dummy

    // simple access functions
    // returns true iff list empty
    Function is_empty() : Bool :=
        return dummy.next == dummy

    // returns pointer to first Item of the list, given list is not empty
    Function first() : *Item :=
        assert (not is_empty())
        return dummy.next

    // returns pointer to last Item of the list, given list is not empty
    Function last() : *Item :=
        assert (not is_empty())
        return dummy.prev

    /* Splice is an all-purpose tool to cut out parts from a list
       Cut out (a, ... b) form this list and insert after t */
    Procedure splice(a, b, t : *Item) :=
        assert (
            b is not before a
            and
            t not between a and b
        )
        // Cut out (a, ... , b)
        a->prev->next := b->next
        b->next->prev := a->prev
```

```
    // insert (a, ... b) after t
    t->next->prev := b
    b->next := t->next
    t->next := a
    a->prev := t


// Moving items by utilising splice
//Move item a after item b
Procedure move_after(a, b: *Item) :=
    splice(a, a, b)

// Move item a to the front of the list
Procedure move_to_front(a: *Item) :=
    move_after(a, dummy)

Procedure move_to_back(a: *Item) :=
    move_after(b, last())

// Deleting items by moving them to a global freeList
// remove item a
Procedure remove(a: *Item) :=
    move_after(b, freeList.dummy)

// remove first item
Procedure pop_front() :=
    remove(first())

//remove last item
Procedure pop_back() :=
    remove(last())

// Inserting Elements
// Insert an item with value x after item a
Function insert_after(x : T, a : *Item) : *Item :=
    checkFreeList() //make sure freeList is non empty
    b := freeList.first() // obtain an item b to hold x
    move_after(b, a) // insert b after a
    b->e := x // set the data item value of b to x
    return b

// Manipulating whole lists
```

```
    Procedure concat(L : List) :=
        splice(L.first(), L.last(), last()) //move whole of L after last
 ↪  element of this list

    Procedure clear()
        freeList.concat(this) //after this operation from from first to last
 ↪  element of this
                              // list are concatenated to the freeList,
 ↪  leaving only the
                              // dummy element in this list.

    Fuction get(i )
```

Splicing

The code for splicing of the `List` class:

```
/* Splice is an all-purpose tool to cut out parts from a list
    Cut out (a, ... b) form this list and insert after t */
Procedure splice(a, b, t : *Item) :=
    assert (
        b is not before a
        and
        t not between a and b
    )
    // Cut out (a, ... , b)
    a->prev->next := b->next
    b->next->prev := a->prev

    // insert (a, ... b) after t
    t->next->prev := b
    b->next := t->next
    t->next := a
    a->prev := t
```

- Dlist cut-out $(a, ..., b)$ (see Figure 2.1):
- Dlist insert $(a, ..., b)$ after t (see Figure 2.2):

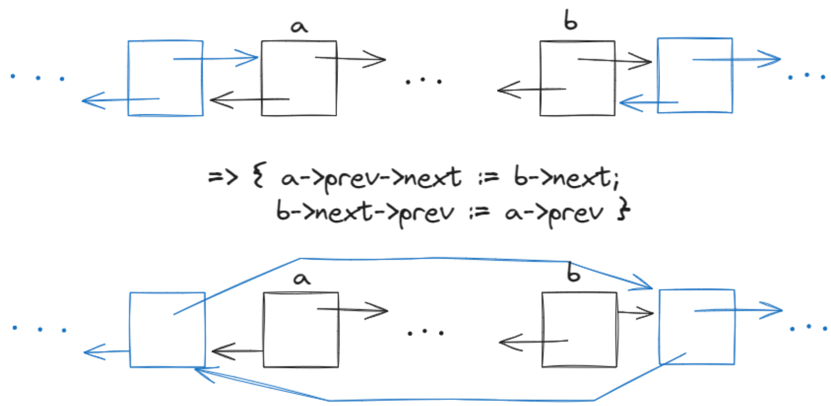


Figure 2.1: cutout

Speicherverwaltung ./.FreeList

Methods (?):

- Niavely: allocate memory for each new element, deallocate memory after deleting each element:
  - advantage: simplicity
  - disadvantage: requires a good implementation of memory management: potentially very slow
- "global" `freeList` (e.g. `static` member in C++)
  - doubly linked list of all not used elements
  - transfer 'deleted' elements in `freeList`.
  - `checkFreeList` allocates, in case the list is empty

Real implementations: * naiv but with well implemented, efficient memory management * refined Free List Approach (class-agnostic, release) * implementation-specific.

Deleting Elements

Deleting elements realised by moving them to the global `freeList`:

```
Procedure remove(a: *Item) :=
    move_after(a, freeList.dummy) // item a is now a 'free' item.

Procedure pop_front() :=
    remove(first())
```

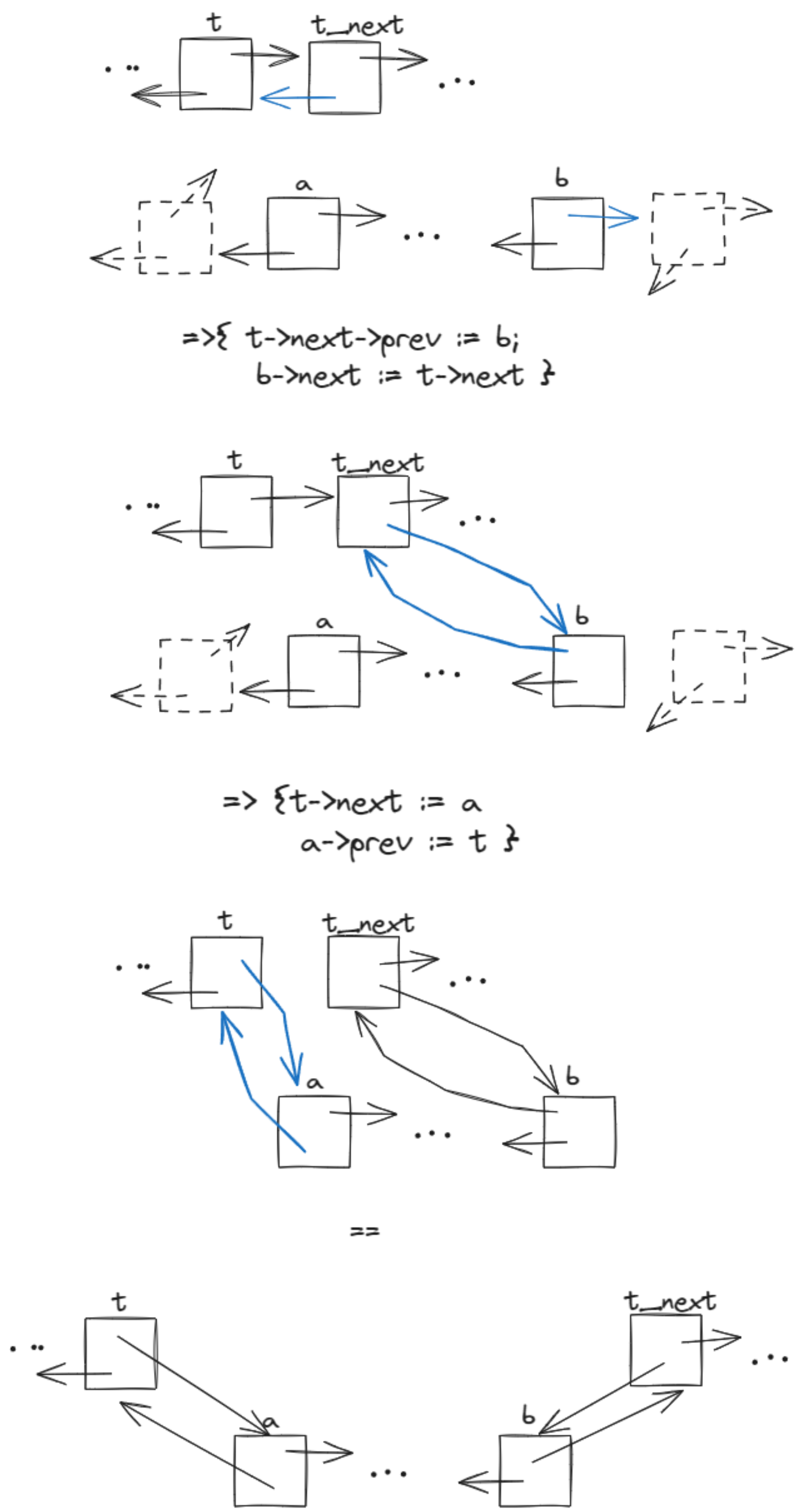


Figure 2.2: insert

```
Procedure pop_back() :=
    remove(last())
```

Inserting Elements

Inserting elements into a list l also utilizes `freeList`, by fetching its first element an moving it into l.

```
Function insert_after(x : T, a : *Item) : *Item :=
    checkFreeList() //make sure freeList is non empty
    b := freeList.first() // obtain an item b to hold x
    move_after(b, a) // insert b after a
    b->e := x // set the data item value of b to x
    return b

Function insert_before(x : T, b : *Item) : *Item :=
    return insert_after(x, b->prev)

Procedure push_front(x : T) :=
    insert_after(x, dummy)

Procedure push_back(x : T) :=
    insert_after(x, last())
```

Manipulating whole Lists

```
// Manipulating whole lists
Procedure concat(L : List) :=
    splice(L.first(), L.last(), last()) //move whole of L after last element
 ↪  of this list

Procedure clear()
    freeList.concat(this) //after this operation from from first to last
 ↪  element of this
                            // list are concatenated to the freeList, leaving
 ↪  only the
                            // dummy element in this list.
```

This operations require **constant time** - indeendent of the list size!

3 Arrays

An array is a contigious sequence of memory cells.

3.1 Bounded Arrays

Bounded arrays have fixed size and are an efficient data structure.

- Size must be known during compile time and is fixed.
- Its memory location in the stack allows many compiler optimizations.

3.2 Unbounded Arrays

The size of an **unbounded array** can dynamically change during run-time. From the user POV it provides the same behaviour as a linked list.

It allows the operations:

- `pushBack(e: T)`: insert an element at the end of the array
- `popBack(e: T)`: remove an element at the end of the array

3.2.1 Memory Management

- `allocate(n)`: request a n contigious blocks of memory words and returns the address value of the first block. This we have the memory blocks:

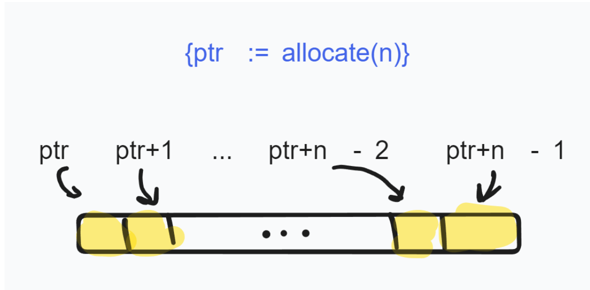


Figure 3.1: array memory allocation

where `ptr + i` addresses are determined by pointer arithmetic.

- `dispose(ptr)` marks the memory address value held in `ptr` as free, effectively deleting the object held there.

In general, the allocated memory can't grow dynamically during life time, since the immediate memory block after the last one might get unpredictably occupied $\Rightarrow$ If we need a new memory block of size $n' > n$, we must allocate a new block, copy the old block contents, and finally free it.

3.2.2 Implementation

First we consider a slow variant:

```
Class UArraySlow<T>:=
  c := 0 : Nat // capacity
  b : Array[0..c-1]<T> // the array itself

  pushBack(el : T) : void :=
    // c++
    // allocate new array on heap with new capacity
    // copy elements over from the old array
    // insert el at the last location

  popBack() : void :=
    // analagous
```

Problem: n `pushBack` operations require $1 + \dots + n \in \mathcal{O}(n^2)$ time $\Rightarrow$ slow.

Solution:

Unbounded Arrays with Extra Memory

Idea: Request more memory than initial capacity. Reallocate memory only when array gets full or too empty:

Algorithm design principle: make common case fast.

```
Class UArray<T> :=
  c := 1 : Nat // capacity
  n := 0 : Nat // number of elements in the array

  //invariant n <= c < k*n || (n == 0 && c < 2)
  b : Array[0..c-1]<T>

  // Array access
  Operator [i : Nat] : T :=
    assert(0 <= i < n)
    return b[i]

  // accessor method for n
  Function size() : Nat := return n

  Procedure pushBack(e : T) :=
    if n == c :
      reallocate(2*n) // see definition below
    b[n] := e
    n++

  // reallocates a new memory with a given capacity c_new
  Procedure reallocate(c_new : Nat) :=
    c := c_new
    b_new := new Array[0..c_new - 1]<T>
    //copy elements over to new array
    for (i = 1 to n - 1) :
      b_new[i] := b[i]
    dispose(b)
    b := b_new

  Procedure popBack() :=
    // don't do anything for empty arrays
    assert n > 0
    n--
```

```
if 4*n <= c && n > 0 :
  reallocate(2*n)
```

4 Sorting and Priority Queues

4.1 Sorting Algorithms

4.1.1 Insertion Sort

```
def insertion_sort(a) :
    n = len(a)
    # i = 1
    # sorted a[0..i-1]
    for i in range(1, n) :
        # insert i in the right position
        j = i - 1
        el = a[i]
        while el < a[j] and j > 0 :
            a[j + 1] = a[j]
            j = j - 1
        # el >= a[j] or j == 0
        if el < a[j] : # j == 0
            a[1] = a[0]
            a[0] = el
        else : # el >= a[j]
            a[j + 1] = el
    return a
```

testing insertion sort for some inputs:

```
import numpy as np
for i in range (2, 8) :
    randarr = np.random.randint(1, 20, i)
    print("in:  ", randarr)
    print("out: ", insertion_sort(randarr))
```

```
in:   [19  8]
out:  [ 8 19]
in:   [12  6  1]
```

```
out:   [ 1  6 12]
in:    [2 3 3 8]
out:   [2 3 3 8]
in:    [11  4 18 16 14]
out:   [ 4 11 14 16 18]
in:    [11  4 18  9 13  3]
out:   [ 3  4  9 11 13 18]
in:    [ 8  6  4 11  3 10  7]
out:   [ 3  4  6  7  8 10 11]
```

Following illustrates the state after each insertion (ith iteration):

```python
def insertion_sort_print(a) :
    n = len(a)
    # i = 1
    # sorted a[0..i-1]
    for i in range(1, n) :
        # insert i in the right position
        j = i - 1
        el = a[i]
        while el < a[j] and j > 0 :
            a[j + 1] = a[j]
            j = j - 1
        # el >= a[j] or j == 0
        if el < a[j] : # j == 0
            a[1] = a[0]
            a[0] = el
        else : # el >= a[j]
            a[j + 1] = el
        print("after insertion ", i, ": ", a)
    # return a

a = np.random.randint(-20, 20, 8)
print("input:                ", a)
insertion_sort_print(a)
```

```
input:                 [ 10  15  14   3  11  12 -15 -14]
after insertion  1 :  [ 10  15  14   3  11  12 -15 -14]
after insertion  2 :  [ 10  14  15   3  11  12 -15 -14]
after insertion  3 :  [  3  10  14  15  11  12 -15 -14]
after insertion  4 :  [  3  10  11  14  15  12 -15 -14]
after insertion  5 :  [  3  10  11  12  14  15 -15 -14]
after insertion  6 :  [-15   3  10  11  12  14  15 -14]
```

```
after insertion  7 :  [-15 -14   3  10  11  12  14  15]
```

4.1.2 Selection Sort

4.1.3 Bubble Sort

4.1.4 Merge Sort

Given by the following python implementation:

```python
def merge(a, b) :
    # assert: a and b are sorted
    c = []
    n1 = len(a)
    n2 = len(b)
    k1 = 0
    k2 = 0
    i = 0
    # invariant: merged a[0..k1 - 1] with b[0..k2 - 2]
    while k1 < n1 and k2 < n2 :
        if a[k1] <= b[k2] :
            c.append(a[k1])
            k1 = k1 + 1
        else :
            c.append(b[k2])
            k2 = k2 + 1
    # k1 >= n1 or k2 >= n2
    if k1 == n1 :
        while k2 < n2 :
            c.append(b[k2])
            k2 = k2 + 1
    if k2 == n2 :
        while k1 < n1 :
            c.append(a[k1])
            k1 = k1 + 1
    return c

def merge_sort(a) :
    if len(a) == 1 : return a[0:1]
    n = len(a)
    a1 = a[0 : n // 2]
```

```
    a2 = a[n // 2 : ]
    return  merge(merge_sort(a1), merge_sort(a2))
```

We test on some arrays:

```
for i in range (2, 8) :
    randarr = np.random.randint(-20, 20, i)
    print("in:  ", randarr)
    print("out: ", merge_sort(randarr))
```

```
in:   [-7 -7]
out:  [-7, -7]
in:   [  7  -8 -19]
out:  [-19, -8, 7]
in:   [-11 -12  17  -2]
out:  [-12, -11, -2, 17]
in:   [  6  -6   0  -6 -16]
out:  [-16, -6, -6, 0, 6]
in:   [ 11  -7 -19 -17 -12  13]
out:  [-19, -17, -12, -7, 11, 13]
in:   [  4  15 -14 -19  -6 -14  14]
out:  [-19, -14, -14, -6, 4, 14, 15]
```

4.1.5 Quick Sort

Naively:

```
def quicksort(s) :
    if len(s) <= 1 : return s
    p = s[len(s) // 2]
    a = []
    b = []
    c = []
    for i in range(0, len(s)) :
        if s[i] < p : a.append(s[i])
    for i in range(0, len(s)) :
        if s[i] == p : b.append(s[i])
    for i in range(0, len(s)) :
        if s[i] > p : c.append(s[i])
    return quicksort(a) + b + quicksort(c)
```

testing this naive implementation for some arrays:

```python
for i in range (2, 8) :
    randarr = np.random.randint(-10, 20, i)
    print("in:  ", randarr)
    print("out: ", quicksort(randarr))
```

```
in:   [19  6]
out:  [6, 19]
in:   [ 3 19 -4]
out:  [-4, 3, 19]
in:   [14 -8 13 10]
out:  [-8, 10, 13, 14]
in:   [-6 17 -5 14 -4]
out:  [-6, -5, -4, 14, 17]
in:   [ 4  4  1  4 12 -1]
out:  [-1, 1, 4, 4, 4, 12]
in:   [-7 -4 17 15 16 -8 14]
out:  [-8, -7, -4, 14, 15, 16, 17]
```

4.2 Priority Queues and Heap Data Structure

A set M of Elements $e : T$ with Keys supporting two operations:

- `insert(e)`: Insert e into M.
- `delete_min()`: remove the min element from M and return it.

4.2.1 Applications

- Greedy algorithms (selecting the optimal local optimal solution)
- Simulation of discrete events
- branch-and-bound search
- time forward processing.

4.2.2 Binary Heaps

Heap Property:

- For any leaf $a \in M$ a is a heap.
- Let T_1, T_2 be heaps. If $a \leq x$, $\forall x \in T_1, T_2$, then $T_1 \circ a \circ T_2$ is also a heap.

Complete Binary Tree:

- A **complete** binary tree is a binary tree in which ever lebel, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible.

Heap:

- A **heap** is a complete binary tree that satisfies the heap property:
- A heap can be succinctly represented as an array:

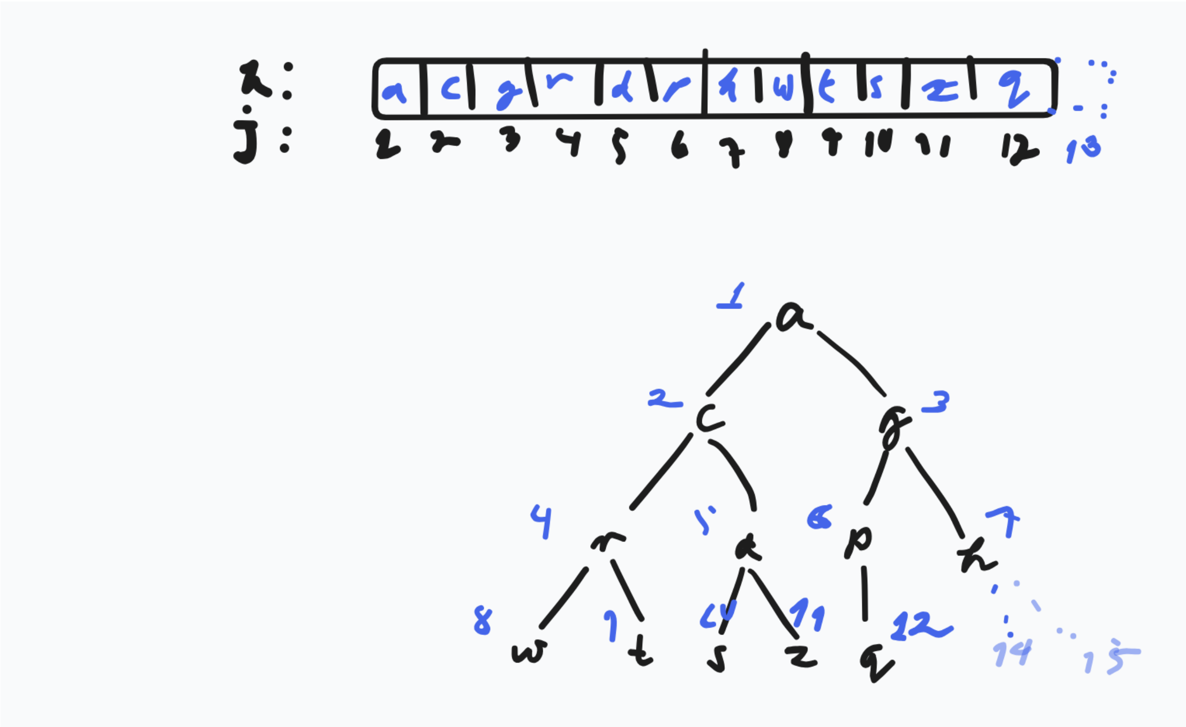


Figure 4.1: heap

- Array `h[1..n]`
- for any given node with the number `j`:
  - left child: `2*j`
  - right child `2*j + 1`
  - parent: `bottom(j/2)`

Pseudocode:

```
Class BinaryHeapPQ(capacity: Nat)<T> :=
    h : Array[1..capacity]<T>
    size := 0 : Nat // current amount of elements
```

```
// Heap-property
// invariant: h[bottom(j/2) <= h(j)], for all j == 2..n

Function min() :=
    assert size > 0 // heap non-emtpy
    return h[1]

Procedure insert(e : T) :=
    assert size < capacity
    size++
    h[n] := e
    siftUp(n)

Procedure siftUp(i : Nat) :=
    // assert Heap-property violated at most at position i
    if i == 1 or h[bottom(i / 2)] <= h[i] then return
    swap(h[i], h[bottom(i/2)])
    siftUp(bottom(i/2))

Procedure popMin : T :=
    result = h[1] : T
    h[1] := h[n]
    n--
    siftDown(1)
    return result

Procedure siftDown (i : Nat) :=
    // assert: Heap property is at most at position 2*i or 2*i + 1
↪ violated
    if 2i > n then return // i is a leaf

    // select the appropriate child
    if 2*i + 1 > n or h[2*i] <= h[2*i + 1] :
    //no right child exists or left child is smaller than right
        m := 2*i
    else : m := 2*i + 1
    siftDown(m)

Procedure buildHeap(a[1..n]<T>) :=
    h := a
    buildRecursive(1)

Procedure buildHeapRecursive(i : Nat) :=
```

```
        if 4*i <= n : // children are not leaves
            buildHeapRecursive(2*i) // assert: heap property holds for left
↪   subtree
            buildHeaprecursive(2*i + 1) // assert: heap property holds for
↪   right subtree
        siftDown(i) //assert Heap property holds for subtree starting at i

    //alternatively
    Procudure buildHeapBackwards :=
        for i := n/2 downto 1 :
            siftDown(i)

    Procedure heapSort(a[1..n]<T>) :=
        buildHeap(a) // O(n)
        for i := n downto 2 do :
            h[i] := deleteMin(); // O(log(n))
```

Heap Insert

```
Procedure insert(e : T) :=
    assert size < capacity
    size++
    h[n] := e
    siftUp(n)

Procedure siftUp(i : Nat) :=
    // assert Heap-property violated at most at position i
    if i == 1 or h[bottom(i / 2)] <= h[i] then return
    swap(h[i], h[bottom(i/2)])
    siftUp(bottom(i/2))
```

Illustration of heap insert:

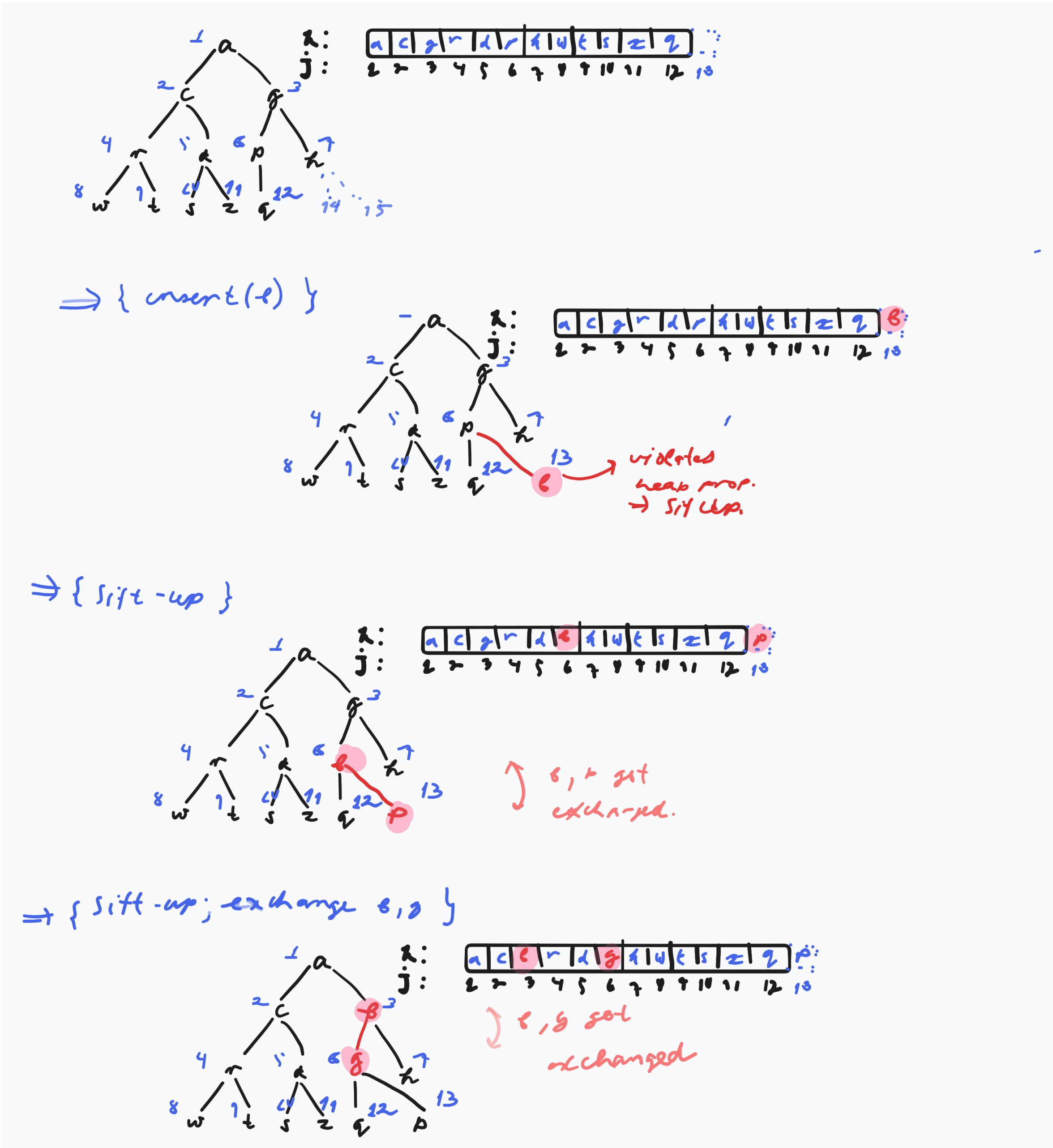


Figure 4.2: heap insert

Heap Pop Min (or Delete Min)

```
 Procedure popMin : T :=
        result = h[1] : T
        h[1] := h[n]
        n--
        siftDown(1)
        return result

Procedure siftDown (i : Nat)
    // assert: Heap property is at most at position 2*i or 2*i + 1 violated
    if 2i > n then return // i is a leaf

    // select the appropriate child
    if 2*i + 1 > n or h[2*i] <= h[2*i + 1] :
    //no right child exists or left child is smaller than right
        m := 2*i
    else : m := 2*i + 1
    siftDown(m)
```

Illustration of pop min:

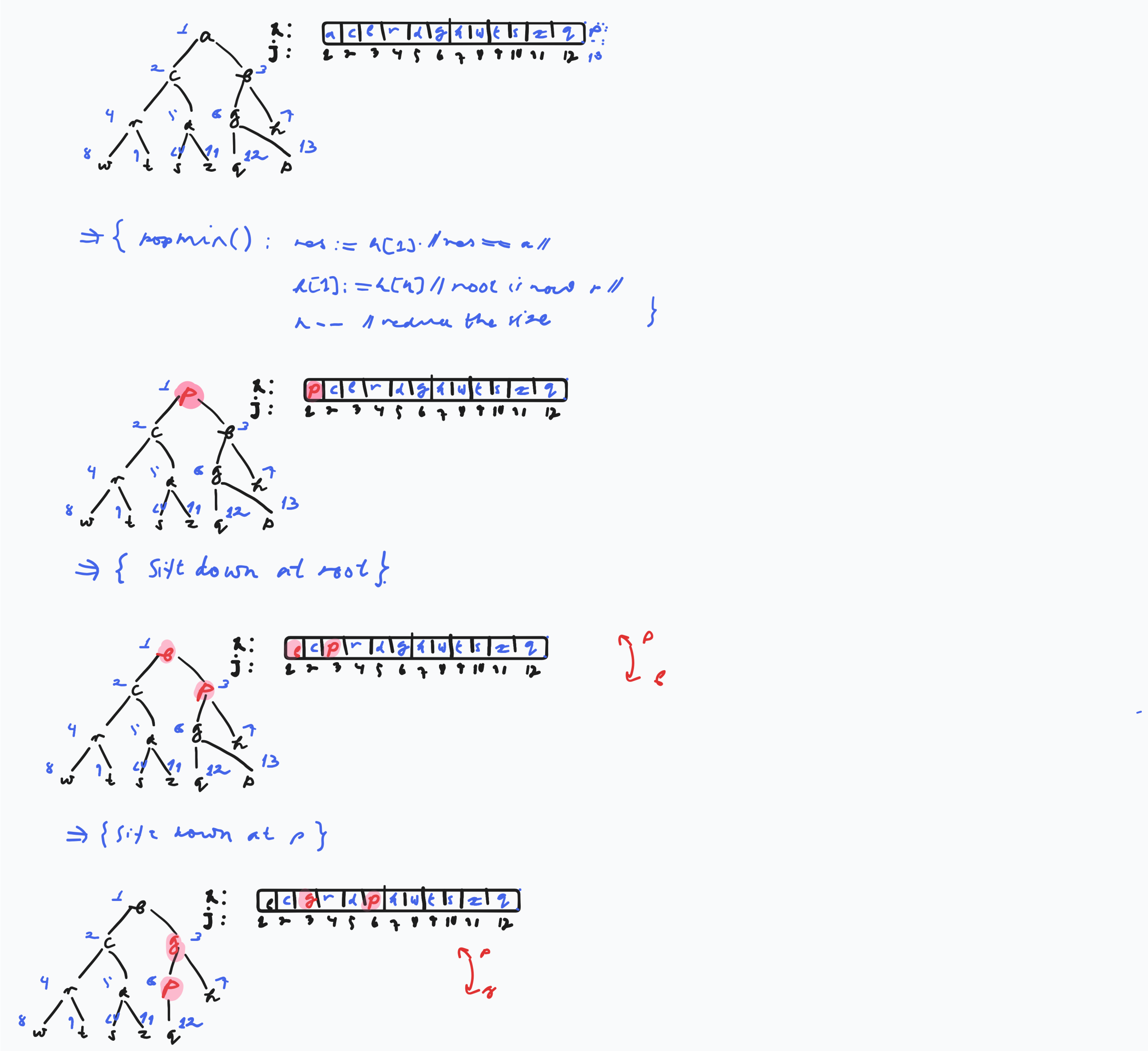


Figure 4.3: heap pop min

Construction of a Binary Heap

- Given are n numbers. Construct a heap from these numbers
- **Naive Solution**: n calls to `insert()` $\Rightarrow \mathcal{O}(n \log(n))$
  - Problem: If numbers are given in an array, we can't perform the construction in

    place.
  - It is slow

- we can do faster and in place in $\mathcal{O}(n)$ time.

Pseudocode for recursive implementation:

```
 Procedure buildHeap(a[1..n] : T) :=
        h := a
        buildRecursive(1)

Procedure buildHeapRecursive(i : Nat) :=
    if 4*i <= n : // children are not leaves
        buildHeapRecursive(2*i) // assert: heap property holds for left
 ↪  subtree
        buildHeaprecursive(2*i + 1) // assert: heap property holds for right
 ↪  subtree
    siftDown(i) //assert Heap property holds for subtree starting at i
```

A simpler iterative one-liner:

```
 Procudure buildHeapBackwards :=
        for i := n/2 downto 1 :
            siftDown(i)
```

$\lfloor i/2 \rfloor$ is the last non-leaf node.

Time complexity of these binary heap construction algorithms is $\boldsymbol{\mathcal{O}}(n)$.

Heapsort

```
Procedure heapSort(a[1..n]<T>) :=
        buildHeap(a) // O(n)
        for i := n downto 2 do :
            h[i] := deleteMin(); // O(log(n))
```

Sorts in decreasing order in $\mathcal{O}(n \log(n))$, by removing the minimal element and writing the return value to the end of the array in place.