Elementary data structures
— T.05 —

Summary:

- Data containers
- Arrays (and circular buffers)
- Linked lists (singly- and doubly-linked)
- Stacks
- Queues
- Deques
- Heaps
- Priority queues
- Binary trees
- Tries
- Hash tables
- Exercises

Recommended bibliography for this lecture:

- **Analysis of Algorithms**, Jeffrey J. McConnell, second edition, Jones and Bertlett Publishers, 2008.
- **Introduction to Algorithms**, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, third edition, The MIT press, 2009.
- **Algorithms**, Robert Sedgewick and Kevin Wayne, fourth edition, Addison Wesley, 2011
- **Estruturas de Dados e Algoritmos em C**, António Adrego da Rocha, terceira edição, FCA.

Data containers

In most programs it is necessary to store and manipulate information. This information may be stored and manipulated directly by the programmer, or it may be encapsulated in a so-called data container. A data container is a data structure that specifies how the information is organized and stored, together with a standardized interface to access and modify the information (thus, a class).

There are many types of data containers, differing on the services they provide and on the computational complexity of these services (the computational complexity depends on how the information is internally organized in the data container). The choice of data container should be done according to the answers to the following questions:

- Does the insertion and retrieval of information obey some rules? (Say, do we always remove the newest, oldest, largest, or smallest item of information?)
- Do we need efficient random access to the information?
- Do we need efficient sequential access to the information?
- Do we need to efficiently insert information in a random location?
- Do we need to efficiently delete information from a random location?
- Do we need to efficiently search for information?
- Is the information single valued, say, an integer, or is it multi-valued, say, a (key,value) pair?

Sometimes we are interested in worst-case costs of an operation. That may be so in an interactive program (such as a video game), because a long operation may give rise to a noticeable delay.

Sometimes we don't care about worst case costs, but are interested only in the average case (or amortized case). That is the case when only the total execution time of a program is important.

Arrays (part 1, computational complexity)

An array is one data structure that can be used to implement a data container. Assuming that the items of information are to be stored consecutively in memory (in array elements) it follows that

- it is necessary to specify the size of the array before it is used; if later on it turns out that that size is too small, it will be necessary to resize the array (that is an $O(n)$ operation, but, if done rarely, its amortized cost is low)
- given a position (an index) random access to information is fast: $O(1)$
- sequential access is also fast
- inserting information at one end of the used part of the array (assuming it is not full) is fast: $O(1)$
- inserting information at an arbitrary location, opening space for it, is slow: $O(n)$
- replacing information at an arbitrary location is fast: $O(1)$
- deleting information at one end is fast: $O(1)$
- deleting information at an arbitrary location, closing the space it would otherwise leave behind, is slow: $O(n)$; without closing the space, it is fast: $O(1)$
- if the information is stored in the array in random order, then searching for information is slow: $O(n)$
- if the information is stored in the array in sorted order, then searching for information is fast: $O(\log n)$

Arrays (part 2, implementation of a circular buffer)

A circular buffer is an array in which index arithmetic is done modulo the size of the array (for an array with n elements, index n is the same as index 0). Besides supporting the usual array operations, by keeping track of the indices of the first and last data items stored in it, a circular buffer also supports $O(1)$ insertion and deletion of data at either end. Thus a circular buffer is a very efficient way of implementing a queue and a deque (see next slides).

Circular buffers are usually used in device drivers to implement efficiently a queue with a given maximum size, because it does not require any dynamic memory operations (allocation and deallocation of memory).

The following C++ code illustrates one possible way to increment and decrement an index in a circular buffer (of floating point numbers):

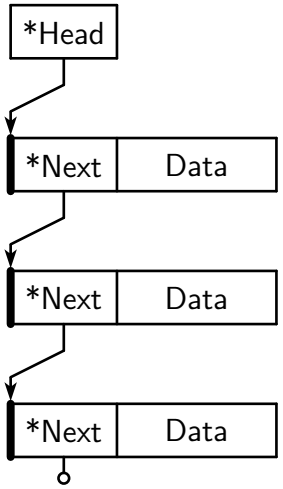
```
class circular_buffer
{
  private:
    int max_size; // the maximum size of the circular buffer
    double *data; // array allocated in the constructor
  public:
    circular_buffer(int max_size = 100)  { this->max_size = max_size; data = new double[max_size]; }
    ~circular_buffer(void)               { delete[] data;                                           }
  private:
    int increment_index(int i)  { return (i + 1 <  max_size) ? i + 1 : 0;           }
    int decrement_index(int i)  { return (i - 1 >= 0       ) ? i - 1 : max_size - 1; }
    //
    // data[i] accesses the number stored in position i
    // data[increment_index(i)], accesses the number stored in the position after position i
    // data[decrement_index(i)], accesses the number stored in the position before position i
    //
    // ... (rest of the code for the circular_buffer class)
    //
}
```

Linked lists (part 1, overview)

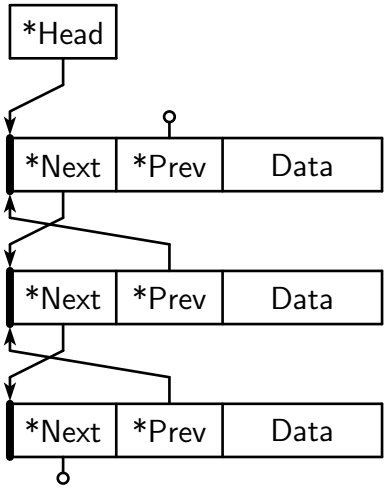
A linked list is a dynamic data structure, because it can grow as much as needed. In order to be able to do this, each node of information contains

- the information itself
- in a singly-linked list, a pointer to the next node of information
- in a doubly-linked list, a pointer to the next node and another to the previous node of information

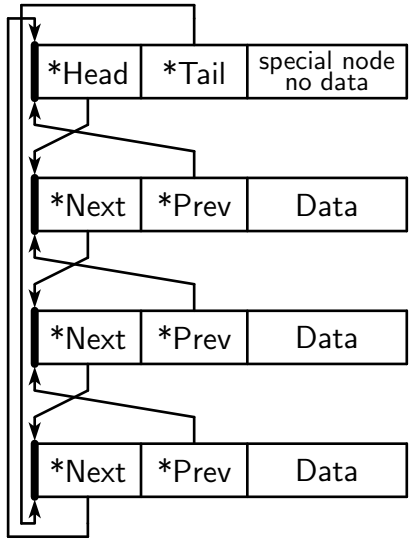
In a singly linked-list, the nodes of information are linked as follows (a small circle represents a `nullptr` pointer, and a `*` before a field name means that it is a pointer):



In a doubly linked-list, the nodes of information are linked as follows (note that when inserting or removing information it may be necessary to deal with `nullptr` pointers):



Using an extra special node to hold the head, in the `Next` field, and the tail, in the `Prev` field, of a doubly linked-list makes its implementation far simpler (no `NULL` or `nullptr` pointers):



(An $*$ denotes a pointer.)

Linked lists (part 2, computational complexity)

Because of the way information is organized in a linked list,

- it is necessary to keep track of the first node (the head) of the list
- it is possible to keep track of the last node (the tail) of the list
- given a position (an index), random access to information is slow: $O(n)$
- forward sequential access (from head to tail) is fast; backward sequential access (from tail to head) is slow for singly-linked lists and fast for doubly-linked lists if the tail is known
- inserting information at the head of the list is fast: $O(1)$
- if the tail of the list is known, inserting information at the end is fast: $O(1)$
- if the tail of the list is not known, inserting information at the end is slow: $O(n)$
- inserting information after a node is fast: $O(1)$
- deleting information at the head of the list is fast: $O(1)$
- on a singly-linked list, deleting a node of information is slow: $O(n)$
- on a doubly-linked list, deleting a node of information is fast: $O(1)$
- searching for information is slow, even if the data is stored in sorted order: $O(n)$

Linked lists (part 3, operations)

The following operations are usually supported in a linked list implementation:

- creation of the linked list
- destruction of the linked list
- insertion of a new node of information (before the head of the list, after the tail of the list, or after a given node of information)
- deletion of a node of information (of the head of the list, of the tail of the list, or of a given node of information)
- given a node of information, determine its next node of information
- given a node of information, determine its previous node of information

Given the class

```
class list_node
{
  private:
    list_node *next;
    // ... (rest of the code for the list_node class)
}
```

the following code finds the tail of a singly-linked list:

```
list_node *tail = head;
if(tail != nullptr)
  while(tail->next != nullptr)
    tail = tail->next;
```

Stacks

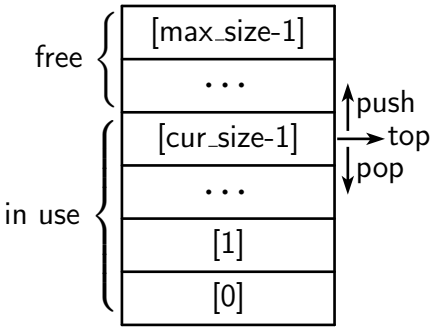
A stack, associated with a usage policy of First In Last Out (FILO), or, what is the same, Last In First Out (LIFO), is a data container that supports the following operations:

- creation of the stack
- destruction of the stack
- add a new element to the top of the stack, called **push**
- remove the element at the top of the stack, called **pop**
- take a look at the top element of the stack, called **top**
- determine the current size of the stack

It is possible to implement a stack using

- an array (in this case the stack has a maximum size, specified when the stack is created),
- a linked list (keeping the top of the stack at the head of the list), or
- a deque (see next slides).

The simplest implementation uses an array and two integers: `max_size`, which is the maximum size of the stack, and `cur_size`, which is the current size of the stack. In this case, the index of the top of the stack is `cur_size-1`, the stack is empty when `cur_size==0` and it is full when `cur_size==max_size`. The following figure illustrates how the information is organized when a stack is implemented using an array.



[**Homework:** implement a stack using a linked list.]

Queues

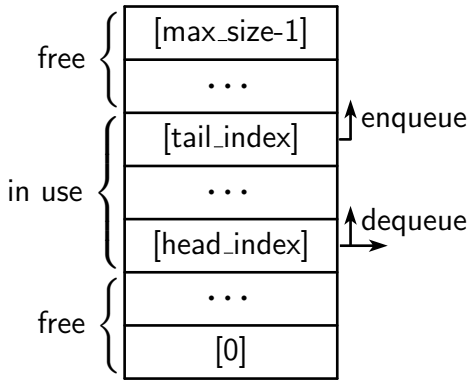
A queue, associated with a usage policy of First In First Out (FIFO), or, what is the same, Last In Last Out (LILO), is a data container that supports the following operations:

- creation of the queue
- destruction of the queue
- add a new element to the back (tail) of the queue, called **enqueue**
- remove the element at the front (head) of the queue, called **dequeue**
- determine the current size of the queue

It is possible to implement a queue using

- a circular buffer (in this case the queue has a maximum size, specified when the queue is created),
- a linked list (preferably one that keeps track of the tail, to make the **enqueue** operation efficient), or
- a deque (see next slides).

The simplest implementation uses a circular buffer and four integers: `max_size`, which is the maximum size of the queue, `cur_size`, which is the current size of the queue, `head_index`, which is the index of the head of the queue, and `tail_index`, which is the index of the tail of the queue.



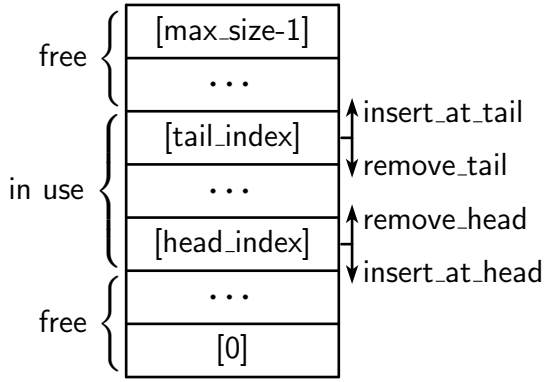
In an empty queue `cur_size==0` and the tail index has to be one more than the head index (modulo `max_size`).

Deques

A deque is a double-ended queue. It is similar to a queue, but it is possible to insert and remove elements at both ends of the queue. It supports the following operations:

- creation of the deque
- destruction of the deque
- insert a new element at the front (head) or at the back (tail) of the deque
- remove the element at the front (head) or at back (tail) the of the deque
- determine the current size of the deque

It can be implemented using either a circular buffer or a doubly-linked list (given that it may be necessary to move in either direction, using a singly-linked list to implement a deque is inefficient).



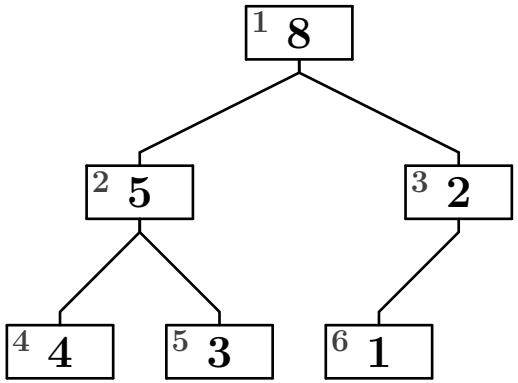
An insertion at either end forces the size of the deque to increase. That determines the direction of movement of the head or of the tail index.

Heaps (part 1, properties)

A heap is an array with internal structure. The internal structure of the array takes the form of an order relation (the heap property) that has to be maintained between the data stored in certain index positions. To be more precise, in a **binary max-heap** with n data items, stored in indices 1, 2, $\ldots$, n, the information stored in index i, with $2 \leqslant i \leqslant n$, cannot be larger than the information stored in index $\lfloor i/2 \rfloor$; $\lfloor x \rfloor$ is the largest integer that is not larger than x (i.e., the `floor` function). For example, the data stored in the following array (left hand side) satisfies the binary max-heap property

index	value
0	-
1	8
2	5
3	2
4	4
5	3
6	1

Implicit binary tree organization



because $8 \geqslant 5$ (index 1 versus index 2), $8 \geqslant 2$ (index 1 versus index 3), $5 \geqslant 4$ (index 2 versus index 4), $5 \geqslant 3$ (index 2 versus index 5), and $2 \geqslant 1$ (index 3 versus index 6). The max-heap property can be easily checked using, for example, the following code

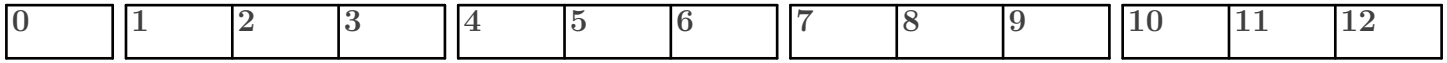
```
for(int i = 2;i <= n;i++)
    assert(heap[i / 2] >= heap[i]);
```

We may also have a binary min-heap, and even multi-way heaps. The starting index is usually either 1 or 0.

Heaps (part 2, m-way heaps)

For each index of a heap we define its parent index, and its children indices. In a max-heap the data stored in a given position cannot be larger that the data stored in the parent position, and it cannot be smaller that the data stored in each one of its children positions. This implicitly defines a tree organization for the data, as shown on the right hand side of the example given in the previous slide. The first index of the array used by the heap is the root index, as it corresponds to the root of the implicit tree.

In a m-way heap with a root index of 0, the parent of index $i > 0$ is $\left\lfloor \frac{i-1}{m} \right\rfloor$ and its children are $mi+1, \ldots, mi+m$. For $m = 3$ the array is implicitly subdivided in the following way:



In a m-way heap with a root index of 1, the parent of index $i > 1$ is $\left\lfloor \frac{i+m-2}{m} \right\rfloor$ and its children are $mi - m + 2, \ldots, mi + 1$. Note that for $m = 2$ the formulas are particularly simple. For $m = 3$ the array is implicitly subdivided in the following way:

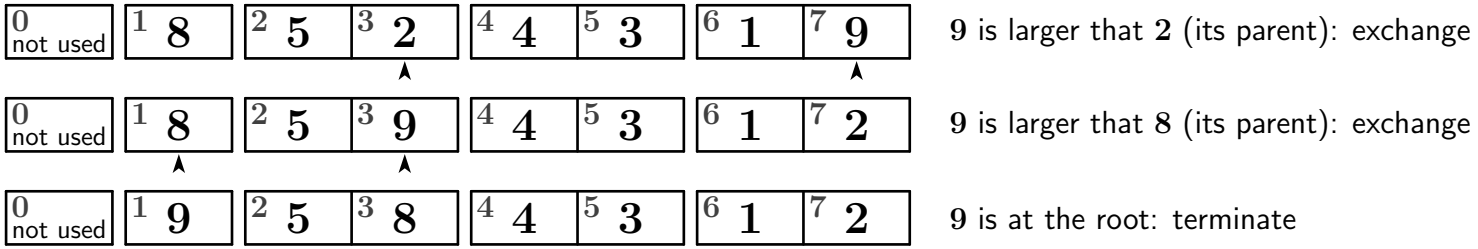
The path to the root of a node with index i is the sequence of indices i, $\mathbf{parent}(i)$, $\mathbf{parent}\big(\mathbf{parent}(i)\big)$, and so on, that stops at the root index. When $m = 2$ and the root index is 1, the path to the root of i is i, $\left\lfloor \frac{i}{2} \right\rfloor$, $\left\lfloor \frac{i}{2^2} \right\rfloor$, $\left\lfloor \frac{i}{2^3} \right\rfloor$, $\ldots$, $\left\lfloor \frac{i}{2^k} \right\rfloor$, where $k = \lfloor \log_2 i \rfloor$; $\log_2 x$ is the base-2 logarithm of x. In particular, for a binary heap of size n, the longest path to the root has length $1 + \lfloor \log_2 n \rfloor$, which is $O(\log n)$.

Heaps (part 3, insertion)

A max-heap supports at least the following operations (a min-heap is similar, with the word "largest" replaced by the word "smallest"):

- creation and destruction of the heap
- inspection of the largest data item (the root of a max-heap holds the largest data value)
- insertion of a data item
- removal of a data item

To insert a data item `v` on a heap with size `n` the first step is to place it at the end of the heap. The heap property is then enforced on the path to the root of the new node, by exchanging the data of a node with that of its parent whenever the heap property is not satisfied (there is no need to change data elsewhere). Thus, insertion is an $O(\log n)$ operation. The following example illustrate what happens when 9 is inserted on the heap $8, 5, 2, 4, 3, 1$:

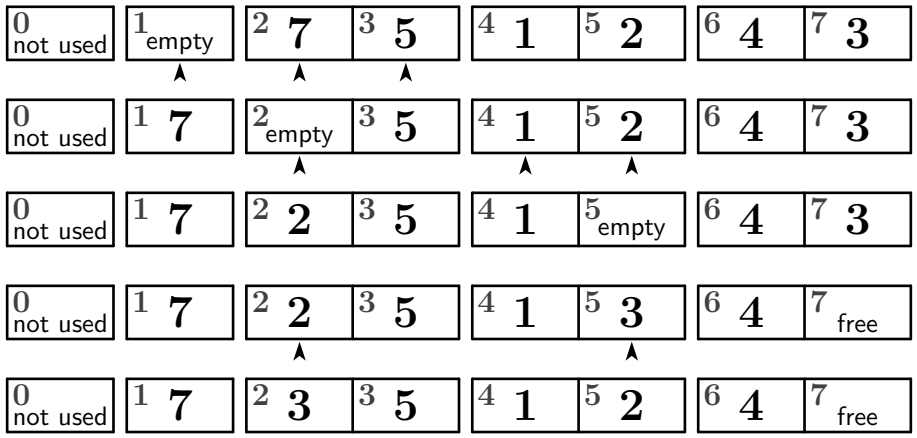


In C or C++, all this can be done as follows ($m = 2$ and the root index is 1):

```
for(i = ++n;i > 1 && heap[i / 2] < v;i /= 2)
  heap[i] = heap[i / 2];
heap[i] = v;
```

Heaps (part 4, removal)

To remove a data item, we replace it by the largest of its children, and then we do the same to fill the empty slot vacated by each child that was moved towards the root, until that cannot be done any more. This procedure leaves an empty slot in the heap. If it is not the last slot, we move the last slot to the empty slot and then enforce the heap property on the path to the root starting at that slot. In the worst case about $2\log_2 n$ operations need to be done — $O(\log n)$ — to remove a data item. The following example illustrates what happens when the root (9) is removed from the heap $9, 7, 5, 1, 2, 4, 3$:



In C or C++, removal of the data at position pos can be done as follows ($m = 2$ and the root index is 1):

```
for(i = pos;2 * i <= n;heap[i] = heap[j],i = j)
  j = (2 * i + 1 <= n && heap[2 * i + 1] > heap[2 * i]) ? 2 * i + 1 : 2 * i; // select largest child
for(;i > 1 && heap[i / 2] < heap[n];i /= 2)
  heap[i] = heap[i / 2];
heap[i] = heap[n--];
```

[**Homework:** what happens in the second `for` loop when i is equal to n?]

Priority queues

A priority queue is a data container that supports the following operations:

- creation and destruction of the priority queue
- inspection of the largest data item (**peek**)
- insertion of a data item (**enqueue**)
- removal of the largest data item (**dequeue**)

It can be implemented easily using a max-heap. Conceptually, a priority queue is similar to an ordinary queue, its only difference being that instead of removing the **oldest** element it is the **largest** that gets removed. (In terms of implementation, a priority queue and an ordinary queue are quite different.)

Instead of inspecting and removing the largest data item, a priority queue can allow the inspection and removal of its smallest data item. This variant of the priority queue is, obviously, implemented using a min-heap.

Binary trees (part 1, overview)

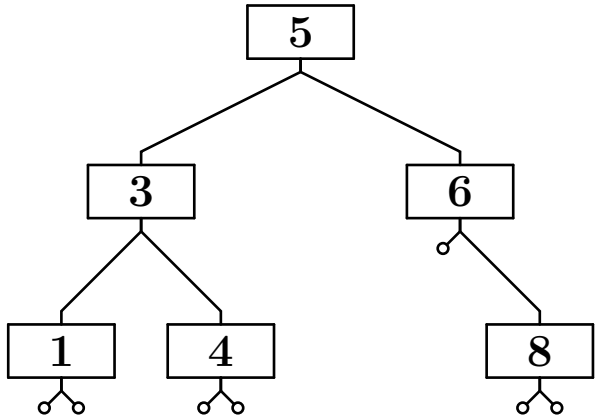
A binary tree is a dynamic data structure composed of nodes. Each node of information contains

- the information itself (the data item)
- a pointer to the node on the "left" side (the left branch); in an ordered binary tree the data items stored on this side are all of them smaller that the data item stored on the node
- a pointer to the node on the "right" side (the right branch); in an ordered binary tree the data items stored on this side are all of them larger that the data item stored on the node
- optionally, a pointer to the parent node

A null pointer (`NULL` in C and `nullptr` in C++) indicates the nonexistence of a node. The **root** node is the node where the tree begins. It is the only node without a parent node. A **leaf** node is a node whose left and right node pointers are null.

The **height** of a tree is the number of levels it has. Each time a left or a right pointer is followed the level increases by 1. It is usual to consider that the root is at level 0. We may also talk about the height of the left or right branches of a node: that it the height of the tree that has as root the left or right node of the node. It is not necessary for a tree to have its leaves all at the same level (but that is usually desired).

Visually, a tree is usually depicted upside-down, i.e., with its root on top:



Binary trees (part 2, node contents)

In these slides each node of the **ordered binary tree** will be implemented (in C) as follows:

```
typedef struct tree_node
{
  struct tree_node *left;   // pointer to the left branch (a sub-tree)
  struct tree_node *right;  // pointer to the right branch (a sub-tree)
  struct tree_node *parent; // optional
  int data;                 // the data item (we use an int here, but it can be anything)
}
tree_node;
```

Each node may also keep information about

- the height of the sub-tree having that node as root,
- the difference of the heights of its left and right branches (the balance)
- the level of the node, or
- other useful information that may be need by a particular program.

In the following slides we present C code snippets of functions that do useful things to a tree. **Study them carefully.** Most of them take the form of a recursive function, because trees are recursive structures: the left and right branches of a node are also trees!

In some of these functions we pass a pointer to the location in memory where the pointer to the root of the tree is stored (a pointer to a pointer). Things are done in this way because it may be necessary to change the root of the tree!

Binary trees (part 3, insertion)

The following non-recursive function creates a new node and inserts it in the ordered tree at the appropriate location:

```
tree_node *new_tree_node(int data,tree_node *parent); // sets left and right to NULL

void insert_non_recursive(tree_node **link,int data)
{
  tree_node *parent = NULL;
  while(*link != NULL)
  {
    parent = *link;
    link = (data <= (*link)->data) ? &((*link)->left) : &((*link)->right); // select branch
  }
  *link = new_tree_node(data,parent);
}
```

This can also be done recursively as follows (the pointers make this much more elegant, but also more difficult to understand):

```
void insert_recursive(tree_node **link,tree_node *parent,int data)
{
  if(*link == NULL)
    *link = new_tree_node(data,parent);
  else if(data <= (*link)->data)
    insert_recursive(&((*link)->left),*link,data);
  else
    insert_recursive(&((*link)->right),*link,data);
}
```

These functions are used as follows:

```
tree_node *root = NULL;

insert_nonrecursive(&root,4);
insert_recursive(&root,NULL,7);
```

What is the height of an initially empty tree after insertion of $1, 2, \ldots, 31$? What about the heigth of an initially empty tree after insertion of $f(1), f(2), \ldots, f(31)$, where the function $f(n)$ reverses the order of the least significant 5 bits of its argument?

Binary trees (part 4, search in an ordered and in an unordered binary tree)

For an **ordered** binary tree, the following non-recursive and recursive functions return one node of the tree for which its data field is equal to a given value (or NULL if there is no such node):

```
tree_node *search_non_recursive(tree_node *link,int data)
{
  while(link != NULL && data != link->data)
    link = (data < link->data) ? link->left : link->right;
  return link;
}

tree_node *search_recursive(tree_node *link,int data)
{
  if(link == NULL || link->data == data)
    return link;
  if(data < link->data)
    return search_recursive(link->left,data);
  else
    return search_recursive(link->right,data);
}
```

These functions are used as follows:

```
tree_node *root,*n;
int data;

n = search_non_recursive(root,data);
n = search_recursive(root,data);
```

For an **unordered** binary tree, the following recursive function returns one node of the tree for which its data field is equal to a given value (or NULL if there is no such node):

```
tree_node *search_recursive(tree_node *link,int data)
{
  node *n;

  if(link == NULL || link->data == data)
    return link;
  // try the left branch
  if((n = search_recursive(link->left,data)) != NULL)
    return n;
  // not found in the left branch, try the right branch
  return search_recursive(link->right,data);
}
```

This function is used as follows:

```
tree_node *root,*n;
int data;

n = search_recursive(root,data);
```

Note that for an unordered binary tree it may be necessary to search the entire tree, which is an order $O(n)$ operation, while in an ordered binary tree it is necessary to examine at most $h + 1$ nodes, where h is the maximum height of any node of the tree.

Binary trees (part 5, traversal)

The following non-recursive and recursive functions traverse the entire tree in different orders:

```
void visit(tree_node *n)
{
  printf("%d\n",n->data);
}

void traverse_breadth_first(tree_node *link)
{
  queue *q = new_queue();
  enqueue(q,link);
  while(is_empty(q) == 0)
  {
    link = dequeue(q);
    if(link != NULL)
    {
      visit(link);
      enqueue(q,link->left);
      enqueue(q,link->right);
    }
  }
  free_queue(q);
}
```

```
void traverse_depth_first_recursive(tree_node *link)
{
  if(link != NULL)
  {
    visit(link);
    traverse_depth_first_recursive(link->left);
    traverse_depth_first_recursive(link->right);
  }
}

void traverse_in_order_recursive(tree_node *link)
{
  if(link != NULL)
  {
    traverse_in_order_recursive(link->left);
    visit(link);
    traverse_in_order_recursive(link->right);
  }
}
```

They are used as follows:

```
tree_node *root;

traverse_breadth_first(root);
traverse_depth_first_recursive(root);
traverse_in_order_recursive(root);
```

[**Homework:** do the `traverse_depth_first` function in a non-recursive way.]

Binary trees (part 5, breadth first traversal, using recursion)

Homework: the `breadth_first` function of this slide can be used to traverse a binary tree in breadth first order. It uses a recursive function to do the actual traversal. Study how they work.

```
int recursive_breadth_first(tree_node *link,int desired_depth)
{
  if(link == NULL)
    return 0; // no node
  if(desired_depth == 0)
  {
    visit(link);
    return 1; // found a node at the desired depth
  }
  return recursive_breadth_first(node->left,desired_depth - 1) | recursive_breadth_first(node->right,desired_depth - 1);
}

void breadth_first(tree_node *root)
{
  for(int desired_depth = 0;recursive_breadth_first(root,desired_depth) != 0;desired_depth++)
    ;
}
```

In particular:

- What is the purpose of the `desired_depth` argument of the recursive function?
- What is the purpose of the value returned by the recursive function?
- Are these functions better than the `traverse_breadth_first` function the previous slide with respect to i) memory consumption, ii) execution time?

Binary trees (part 6a, misc)

Does the purpose of each of the following functions match its name?

```
// use count_nodes(root) to count the nodes of an entire tree

int count_nodes(tree_node *link)
{
  return (link == NULL) ? 0 : count_nodes(link->left) + 1 + count_nodes(link->right);
}


// use count_leaves(root) to count the leaves (terminal nodes) of an entire tree

int count_leaves(tree_node *link)
{
  if(link == NULL)
    return 0;
  if(link->left == NULL && link->right == NULL)
    return 1;
  return count_leaves(link->left) + count_leaves(link->right);
}


// use check_node(root,NULL,INT_MIN,INT_MAX) to check an entire ordered binary tree

void check_node(tree_node *link,tree_node *parent,int min_bound,int max_bound)
{
  if(link != NULL)
  {
    assert(min_bound <= link->data && link->data <= max_bound && link->parent == parent);
    check_node(link->left,link,min_bound,link->data);
    check_node(link->right,link,link->data,max_bound);
  }
}
```

Binary trees (part 6b, misc)

Does the purpose of each of the following functions match its name?

```
// use set_level_nodes(root,0) to set the level of each node of an entire tree

void set_level(tree_node *link,int level)
{
  if(link != NULL)
  {
    link->level = level;
    set_level(link->left,level + 1);
    set_level(link->right,level + 1);
  }
}


// use set_height(root) to set the height of each node of an entire tree

int set_height(tree_node *link)
{
  int left_height,right_height;

  if(link == NULL)
    return 0;
  left_height = set_height(link->left);
  right_height = set_height(link->right);
  link->height = (left_height >= right_height) ? 1 + left_height : 1 + right_height;
  return link->height;
}
```

Binary trees (part 7a, balancing)

A binary tree is said to be **perfect** if all its leaves are at the same level. A perfect binary tree with height h has $2^h - 1$ nodes: 2^{h-1} leaves and $2^{h-1} - 1$ internal nodes. Abusing the concept of perfect binary tree, we will say that a binary tree is also perfect if some of the leaves at the last level of the tree are missing (but all other levels are full). Such a tree has a height as small as possible. That is desirable because many operations that modify a binary tree do a number of elementary operations that is proportional to the height of the tree. A perfect binary tree with n nodes has a height equal to $\lceil \log_2(n+1) \rceil$, which is $O(\log n)$.

A binary tree is said to be **balanced** if for each of its nodes the height of its left branch does not differ by more than 1 from the height of its right branch. A balanced binary tree of height h has at least G_h nodes, where $G_0 = 0$, $G_1 = 1$ and, for $n > 1$, $G_n = G_{n-1} + 1 + G_{n-2}$, and it has at most $2^h - 1$ nodes. It can be verified that $G_n = F_{n+2} - 1$, where F_n is the n-th Fibonacci number. It follows that the height of a balanced binary tree is $\Theta(\log n)$. It is possible, using simple $O(\log n)$ operations, to keep a binary tree balanced after the insertion or removal of a node. Searching is also an $O(\log n)$ operation in a balanced binary tree.

A self-balancing tree, also known as an AVL tree (so-called to honor its two Russian inventors: Adelson-Velsky and Landis), remains balanced after insertions and deletions. When a sub-tree becomes unbalanced during an internal operation, it can be balanced using simple local operations (see next slide).

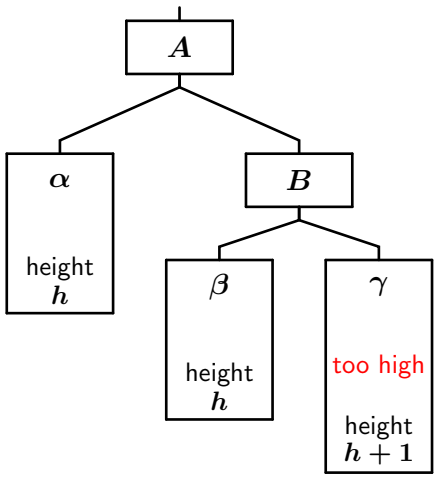
In a computer program, a red-black tree (not studied in this course) can be an alternative to a self-balancing tree. Both have search, insertion and removal worst case costs of $O(\log n)$.

It the data in inserted in an ordered binary tree in **random** order, then the final tree will be, with high probability, be well balanced (meaning that the absolute value of the height difference between the left and right sub-trees of any node will be small).

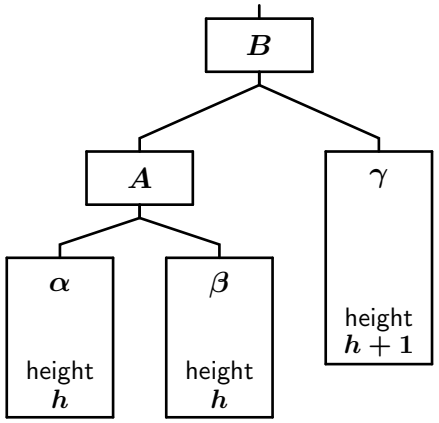
Binary trees (part 7b, how to keep an ordered binary tree balanced)

After insertion of a new node (in sub-tree γ of the following illustration), a sub-tree of a previously balanced and ordered binary tree can become unbalanced in essentially the following two ways (there exist also partial or total mirror images of these two cases, which we do not illustrate here):

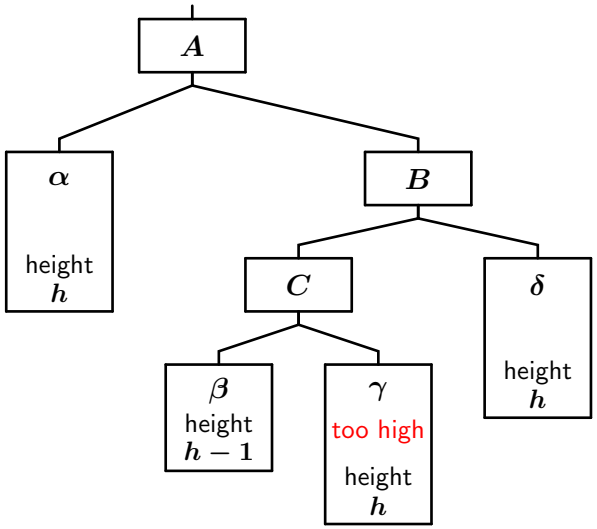
Before:



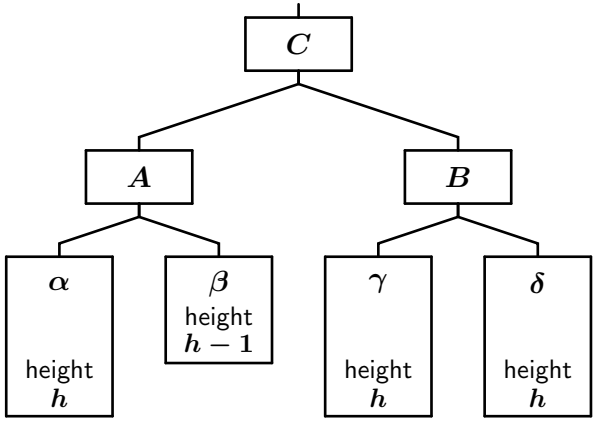
After (single rotation):



Before:



After (double rotation):



Tries

A trie is a prefix tree. Each node of the tree represents a prefix of the information that is to be stored in the tree. This means that it must be possible to split that information into one of more symbols. Each level of the tree stores one more symbol.

When the information to be stored is a concatenation of symbols, each of which can take m different values, a tries is essentially an m-way search tree (each node of the tree can have at most m children.) An index is somnetimes used to select the appropriate children. For example, storing and searching for a telephone number can be done very efficiently using a trie (here m will be 10). The most significant digit of the telephone number selects the child of the root of the tree that must be followed, the second most significant digit selects the child of that node, and so on. Only the nodes that are needed are actually allocated.

TO DO: insert a figure here.

Hash tables (part 1, overview)

A hash table is a data container that supports (at least) the following operations:

- creation and destruction of the hash table
- insertion of a data item composed of two parts: the **key** and its corresponding **value**
- search for a data item with a given key
- removal of a data item given its key

In an array, information is accessed given its index. In a hash table, information is accessed given its key. In a properly sized hash table, the insertion, removal and search operations have very good expected computational complexity: $O(1)$. This is better that the computational complexity of the same operations in a balanced binary tree, which is $O(\log n)$, where n is the number of data items stored in the data container.

A hash table is usually the data container used to implement an associative array, a symbol table, or a dictionary. (Other data containers may also be used for this, but the hash table is usually more efficient.) The key may not be an integer; it may be, for example, a string.

In the insert operation, if a data item with the same key already exists in the hash table, the insert operation should either fail or it should replace the corresponding value (and no new data item is created). The programmer has to decide which is best for a given application.

An hash table is usually implemented using an array. It may be an array of data items (keys and respective values), if **open addressing** (explained later) is used, or it may be an array of pointers to the heads of (doubly-)linked lists of data items, if **chaining** is used. Instead of a pointer to the head of a linked list, it is also possible to use a pointer to the root of a binary tree, or even a pointer to another hash table.

Hash tables (part 2a, hash functions)

Let s be the size of the array used to implement the hash table. An hash function h maps each possible key k to an integer i in the range $0, \ldots, s-1$. This integer will then be used to access the array. This conversion is necessary because the key itself may not be an integer, and even if it is an integer, its value may be too small or too large.

If the number of keys, n, is larger than the size of the array, s, then it is inevitable that two (or more) keys map, via the hash function, to the same index. Even when n is smaller than s, it is possible, if the hash function is not chosen with extreme care, for these so-called **collisions** to happen. Indeed, due to the birthday paradox, if the hash function spreads the indices in an uniform way, then there is at least a 50% chance of a collision when $n \geqslant (1+\sqrt{1+8s\log 2})/2$.

A good hash function should attempt to avoid too many collisions. There are many ways to attempt to do this. One of them it to treat the key, or rather, its memory representation, as a possibly very large integer, and to choose as hash function the remainder of the division of this large integer by the array size. When the key is a string this gives rise to code such as:

```
unsigned int hash_function(const char *str,unsigned int s)
{ // for 32-bit unsigned integers, s should be smaller that 16777216u
  unsigned int h;

  for(h = 0u;*str != '\0';str++)
    h = (256u * h + (0xFFu & (unsigned int)*str)) % s;
  return h;
}
```

It turns out that hash functions of this form are better (less collisions) when s is a prime number. (Furthermore, the order of 256 in the multiplicative group of remainders co-prime to s should be large; in particular, s should not be an even number.)

Hash tables (part 2b, more hash functions)

The hash function presented in the previous slide is reasonably good but it is slow, because it requires a remainder operation in each iteration of the for loop, and it works better when s is a prime number. Furthermore, for a 32-bit `unsigned int` data type, due to a possible integer overflow, it should not be used when `s` is larger than or equal to $2^{32-8} = 16777216$.

One way to solve these problems is to get rid of all but the last of the remainder operations, as done in the following variant of the hash function of the previous slide:

```
unsigned int hash_function(const char *str,unsigned int s)
{
  unsigned int h;

  for(h = 0u;*str != '\0';str++)
    h = 157u * h + (0xFFu & (unsigned int)*str); // arithmetic overflow may occur here (just ignore it!)
  return h % s; // due to the unsigned int data type, it is guaranteed that 0 <= h % s < s
}
```

(The multiplication factor 157 was chosen is an almost arbitrary way.) Note that return values smaller that $2^{32} \bmod s$ (for a 32-bit data type) will be slightly more probable than those larger than or equal to $2^{32} \bmod s$ (and, of course, smaller that s). This defect does not cause any significant problem when s is many times smaller (say, 1000 times smaller) that the largest possible unsigned integer. Of course, this potential problem can be almost eliminated if 64-bit integers are used (`unsigned long long` data type).

Hash tables (part 2c, even more hash functions)

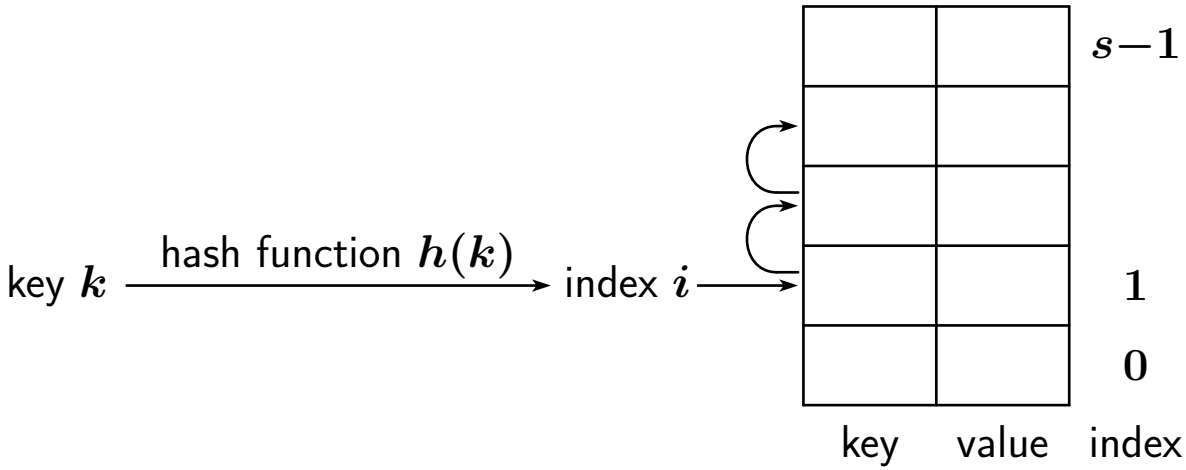
Other possible hash functions are based on so-called cyclic redundancy checksums (CRC), or on so-called message digest signatures (such as MD5), or on secure hash algorithms (such as SHA-1). The following hash function is based on a 32-bit cyclic redundancy checksum.

```
unsigned int hash_function(const char *str,unsigned int s)
{
  static unsigned int table[256];
  unsigned int crc,i,j;

  if(table[1] == 0u) // do we need to initialize the table[] array?
    for(i = 0u;i < 256u;i++)
      for(table[i] = i,j = 0u;j < 8u;j++)
        if(table[i] & 1u)
          table[i] = (table[i] >> 1) ^ 0xAED00022u; // "magic" constant
        else
          table[i] >>= 1;
  crc = 0xAED02020u; // initial value (chosen arbitrarily)
  while(*str != '\0')
    crc = (crc >> 8) ^ table[crc & 0xFFu] ^ ((unsigned int)*str++ << 24);
  return crc % s;
}
```

For the curious, the "magic" constant encodes the coefficients (bits) of a primitive polynomial in the finite field $GF(2^{32})$. In this case the polynomial is $x^{32} + x^{30} + x^{26} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^2 + 1$. [**Homework:** In the function given above replace the 32-bit CRC by a 64-bit CRC; use a "magic" constant of `0xAED0AED0AED0011Full`.]

Hash tables (part 3a, open addressing)

When a hash table uses open addressing the key and respective value are stored directly in the array:



This implies that $n \leqslant s$, and that it is necessary to resolve collisions by looking at other positions of the array when position i, with $i = h(k)$, does not contain the correct key. One possibility is to try $(i+1) \bmod s$, $(i+2) \bmod s$, and so on, until either the desired key is found or an empty array position is found. Instead of trying consecutive positions, it is also possible to try positions further apart, with jumps of j between positions: $(i+j) \bmod s$, $(i+2j) \bmod s$, and so on. For this to work is is necessary and sufficient that $\gcd(j,s) = 1$, which is ensured if $0 < j < s$ and if s is a prime number. Instead of using a fixed j, in **double hashing** each key uses its own j, computed by another hash function.

Open addressing has several major disadvantages:

- the hash table cannot have more that s keys
- when the hash table is nearly full and there are collisions the worst search time can be quite large
- it is difficult to remove keys from the hash table

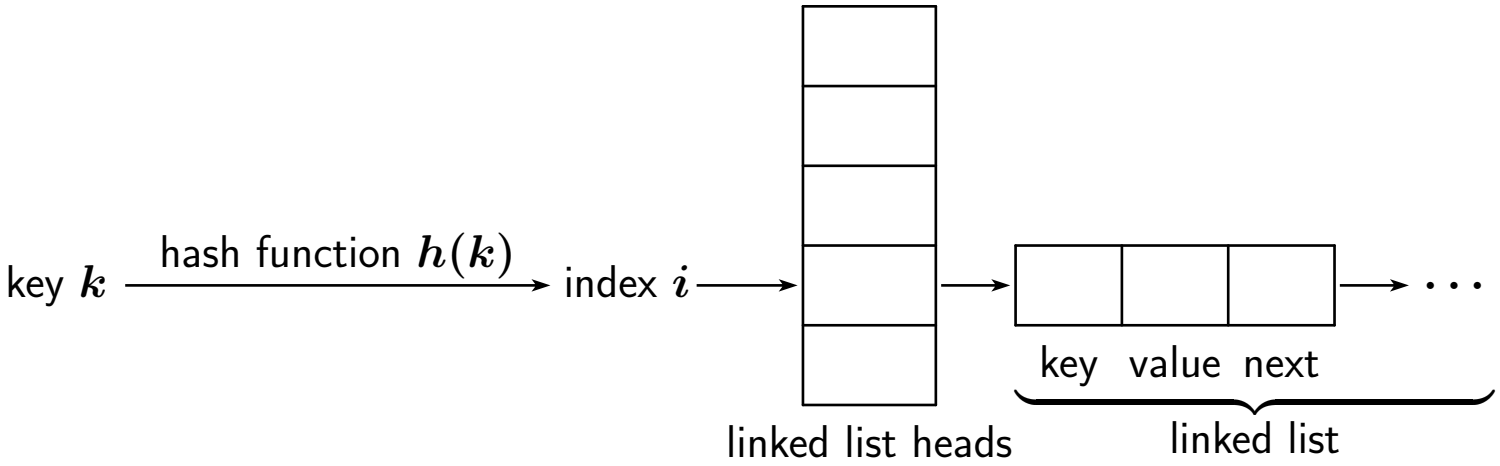
Hash tables (part 3b, open addressing)

The following C code exemplifies one way to perform a key search in a hash table that uses open addressing (compare to similar code that used separate chaining, presented in part 4b):

```
typedef struct hash_data
{
  char key[10]; // empty when key[0] = '\0'
  int value;
}
hash_data;

#define hash_size 1009u

hash_data hash_table[hash_size];

hash_data *find_data(const char *key)
{
  unsigned int idx;

  idx = hash_function(key,hash_size);
  while(hash_table[idx].key[0] != '\0' && strcmp(key,hash_table[idx].key) != 0)
    idx = (idx + 1u) % hash_size; // try the next array position
  return (hash_table[idx].key[0] == '\0') ? NULL : &hash_table[idx];
}
```

Hash tables (part 4a, separate chaining)

When the hash table uses chaining, sometimes also called separate chaining, the array stores pointers to the heads of linked lists. For each key the hash function specifies in which list we will operate (search, insert or remove). Of course, it is possible to replace the linked lists by a more sophisticated data structure, such as a binary search tree (or even another hash table!), that provides the same operations but with lower computational complexity.



When separate chaining is being used, the main purpose of the hash function is to distribute the keys as evenly an possible among the s positions of the array. A good spread implies a small number of collisions, and so a search operation will be fast. (If linked lists are being used, the worst search time is proportional to the length of the longest linked list.)

Because linked lists are dynamic data structures, a hash table that uses them can store more keys that the size of the array. The average search time will be $\mathbf{max}(1, n/s)$ if all keys are equally probable. So, the performance of a hash table implemented with separate chaining will degrade gracefully if its array is under-sized

Hash tables (part 4b, separate chaining)

The following C code exemplifies one way to perform a key search in a hash table that uses separate chaining (compare to similar code that used open addressing):

```
typedef struct hash_data
{
  struct hash_data *next; // for the linked list
  char key[10];
  int value;
}
hash_data;

#define hash_size 1009u
hash_data *hash_table[hash_size];

void init_hash_table(void)
{
  for(unsigned int idx = 0u;idx < hash_size;idx++)
    hash_table[idx] = NULL;
}

hash_data *find_data(const char *key)
{
  unsigned int idx = hash_function(key,hash_size);
  hash_data *hd = hash_table[idx];
  while(hd != NULL && strcmp(key,hd->key) != 0)
    hd = hd->next;
  return hd;
}
```

Hash tables (part 4c, separate chaining)

The following C code exemplifies how to allocate a new `hash_data` structure:

```
#include <stdio.h>
#include <stdlib.h>

hash_data *new_hash_data(void)
{
  hash_data *hd = (hash_data *)malloc(sizeof(hash_data));
  if(hd == NULL)
  {
    fprintf(stderr,"Out of memory\n");
    exit(1);
  }
  return hd;
}
```

The following C code exemplifies how to visit all nodes of a hash table:

```
void visit_all(void)
{
  unsigned int i;
  hash_data *hd;

  for(i = 0u;i < hash_size;i++)
    for(hd = hash_table[i];hd != NULL;hd = hd->next)
      visit(hd);
}
```

Hash tables (part 4d, separate chaining)

The following C code presents a more efficient way to allocate a new hash_data structure:

```
hash_data *free_hash_data = NULL;

hash_data *new_hash_data(void)
{
  if(free_hash_data == NULL)
  {
    const int n_nodes = 100;
    free_hash_data = (hash_data *)malloc((size_t)n_nodes * sizeof(hash_data));
    if(free_hash_data == NULL)
    {
      fprintf(stderr,"Out of memory\n");
      exit(1);
    }
    for(int i = 0;i < n_nodes;i++)
      free_hash_data[i].next = (i + 1 < n_nodes) ? &free_hash_data[i + 1] : NULL;
  }
  hash_data *hd = free_hash_data;
  free_hash_data = free_hash_data->next;
  return hd;
}

void free_hash_data(hash_data *hd)
{
  hd->next = free_hash_data;
  free_hash_data = hd;
}
```

Hash tables (part 5, perfect hash functions)

A perfect hash function is an hash function that does not generate any collisions. Perfect hash functions are desired when the set of keys is **fixed and known** (for example, the reserved keywords of a programming language). In a perfect hash function it is usually desired that $s = n$ (to save space). As there are **no collisions**, the hash table is best implemented using **open addressing**.

Although perfect hash functions are rare (recall the birthday paradox), it is not difficult to construct a perfect hash function $h(k)$, that maps k to one of the integers $0, 1, \ldots, s-1$. Mathematically this can be expressed in a more compact way by $h : k \mapsto \{i\}_{i=0}^{s-1}$. The main idea is to use two distinct hash functions $h_1 : k \mapsto \{i\}_{i=0}^{s-1}$ and $h_2 : k \mapsto \{i\}_{i=0}^{t-1}$ and to use an auxiliary table T of size t. Indeed, when t is not too small ($t \approx s/2$ works well), and assuming that there are no key pairs that give rise to simultaneous collisions in the two hash functions, is it usually possible to construct the table in such a way that

$$h(k) = \big(h_1(k) + T[h_2(k)]\big) \bmod s$$

is a perfect hash function. Note that for a key k without a collision in $h_2(k)$ it is possible to assign to $h(k)$ any desired value.

Let K_i be the set of keys for which h_2 evaluates to i; mathematically we have $K_i = \{\ k\ :\ h_2(k) = i\ \}$. The table T can be constructed using a greedy approach by considering each K_i in turn, starting with the largest set and ending with the smallest set (the size of a set is its number of elements). For each i one tries $T[i] = 0$, $T[i] = 1$, and so on, until either s is reached (a failure!) or all the values of $h(k)$, for $k \in K_i$, do not collide with the values of $h(k)$ for the k belonging to the sets already dealt with. Sets with 1 or 0 elements, done at the end, do not pose any problem! In practice, this greedy approach works surprising well, if one is not too ambitious in the choice of t.