Leibniz Universität Hannover

Programmieren 1

Dynamische Datenstrukturen

Human-Computer Interaction Group

Prof. Dr. Michael Rohs
michael.rohs@hci.uni-hannover.de

Lectures

#	Date	Topic	HÜ→	HÜ←
1	14.10.	Organization, computers, programming, algorithms, PostFix introduction (execution model, IDE, basic operators, booleans, naming)	1	20.10. 23:59
2	21.10.	PostFix (primitive types, functions, parameters, local variables, tests), recipe for atomic data	2	27.10. 23:59
3	28.10.	PostFix (operators, array operations, string operations), recipes for enumerations and intervals	3	3.11. 23:59
4	4.11.	Recipes for compound and variant data, iteration and recursion, PostFix (loops, association arrays, data definitions)	4	10.11. 23:59
5	11.11.	C introduction (if, variables, functions, loops), Programming I C library	5	17.11. 23:59
6	18.11.	Data types, infix expressions, C language (enum, switch)	6	24.11. 23:59
7	25.11.	Compound and variant data, C language (formatted output, struct, union)	7	1.12. 23:59
8	2.12.	C language (arrays, pointers) arrays: fixed-size collections, linear and binary search	8	8.12. 23:59
9	9.12.	Dynamic memory (malloc, free), recursion (recursive data, recursive algorithms)	9	15.12. 23:59
10	16.12.	Linked lists, binary trees, search trees	10	22.12. 23:59
11	13.1.	C language (program structure, scope, lifetime, linkage), function pointers, pointer lists	11	12.1. 23:59
12	20.1.	List and tree operations (filter, map, reduce), objects, object lists	12	19.1. 23:59
13	27.1.	Dynamic data structures (stacks, queues, maps, sets), iterators, documentation tools	(13)	

Preview: Common Data Structures for Programming

- Organizing collections of elements

- Vector
- List
- Stack
- Queue
- Map
- Set
- Tree

Leibniz Universität Hannover

PROGRAMMING WITH DATA STRUCTURES

Data Structures

- Important part of the programmer's toolbox
- Organizing collections of elements
  - Efficient lookup of elements
  - Efficient insertion and removal of elements
  - Maintaining order (e.g., alphabetical)
  - Reflecting the structure of the data (e.g., a hierarchy)
- Different data structures for different tasks
  - Different design goals and purposes
  - Different runtime costs
    - Finding an element in linked list may require inspecting all the elements
  - Different memory requirements
    - Linked list needs memory for storing nodes and for pointers between them

Important Data Structures

- Vector: A dynamic array
- List: Linear sequence of elements
- Stack: Last-in, first-out (LIFO)
- Queue: First-in, first-out (FIFO)
- Map: Associations of keys to values
- Set: Mathematical (finite) set
- Tree: Hierarchy of elements
- Graphs: Networks of elements

Typical Operations on these Data Structures

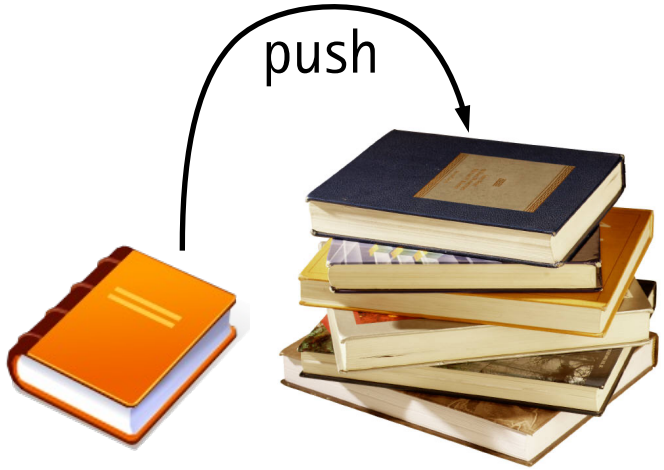
- List: create, prepend, head, tail, remove, is_empty
- Stack: create, push, pop, is_empty
- Queue: create, enqueue (add), dequeue (remove+return), is_empty
- Map: create, put, get, remove
- Set: create, add, remove, contains, is_empty,
  union, intersection, difference, is_subset_of

Leibniz Universität Hannover

STACK, QUEUE

A Stack of Items

- Stack ("Stapel") is a collection that stacks items on top of each other
- Principle: last-in first-out (LIFO)
  - The last item added will be returned when reading
  - Cannot directly access items below top one
- Operations
  - push: add an item on top of stack
  - pop: remove and return item from top
  - peek: look at item on top, do not remove
  - is_empty: check whether there are any items
  - clear: remove all elements from stack



Stack and Queue

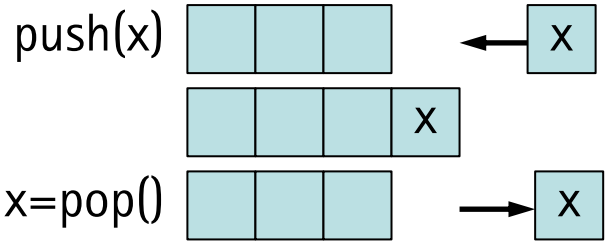
- Stack: Last in first out (LIFO)
  - The element that was added last (push) is returned first (pop)
  - Application examples:
    runtime stack (local variables, recursion),
    stack of books

- Queue: First in first out (FIFO)
  - The element that was added first (put, enqueue) is returned first (get, dequeue)
  - Application examples:
    output buffer (sending data over network)
    waiting at checkout... (Mensa)

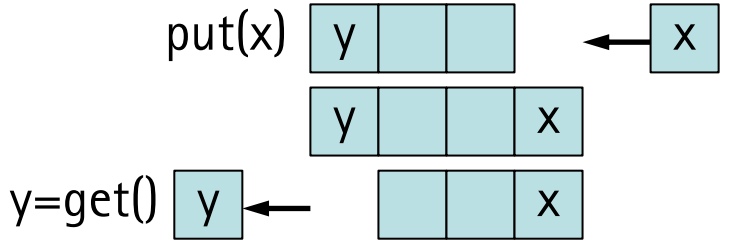
Image source: Getty

Stack and Queue

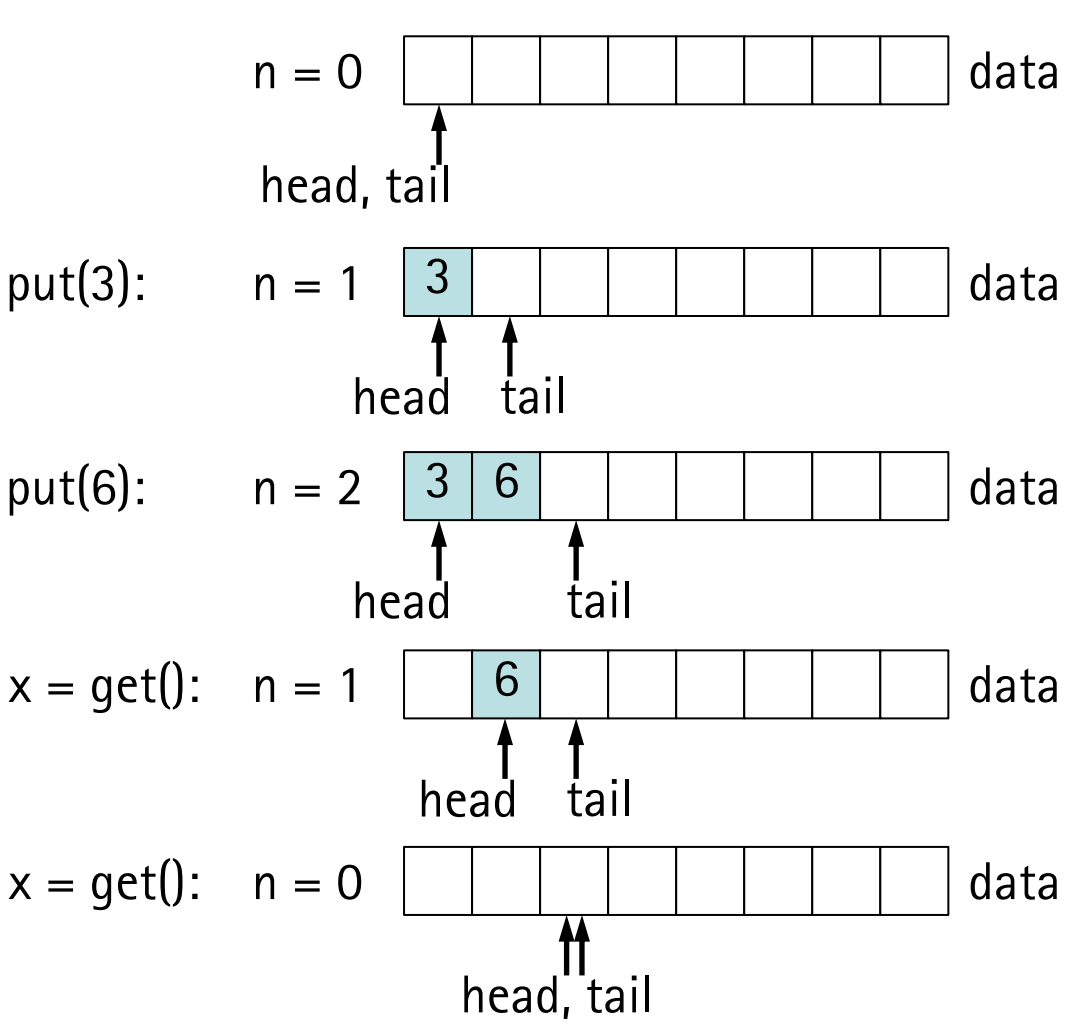
- Stack: last in first out (LIFO)
  - The element that was added last (push) is returned first (pop)
  - push(x); // x is topmost stack element
  - x = pop(); // returns topmost stack element



- Queue: first in first out (FIFO)
  - The element that was added first (put) is returned first (get)
  - put(x); // x is added to the back of the queue
  - y = get(); // y is returned from the front of the queue



Fixed Size Queue



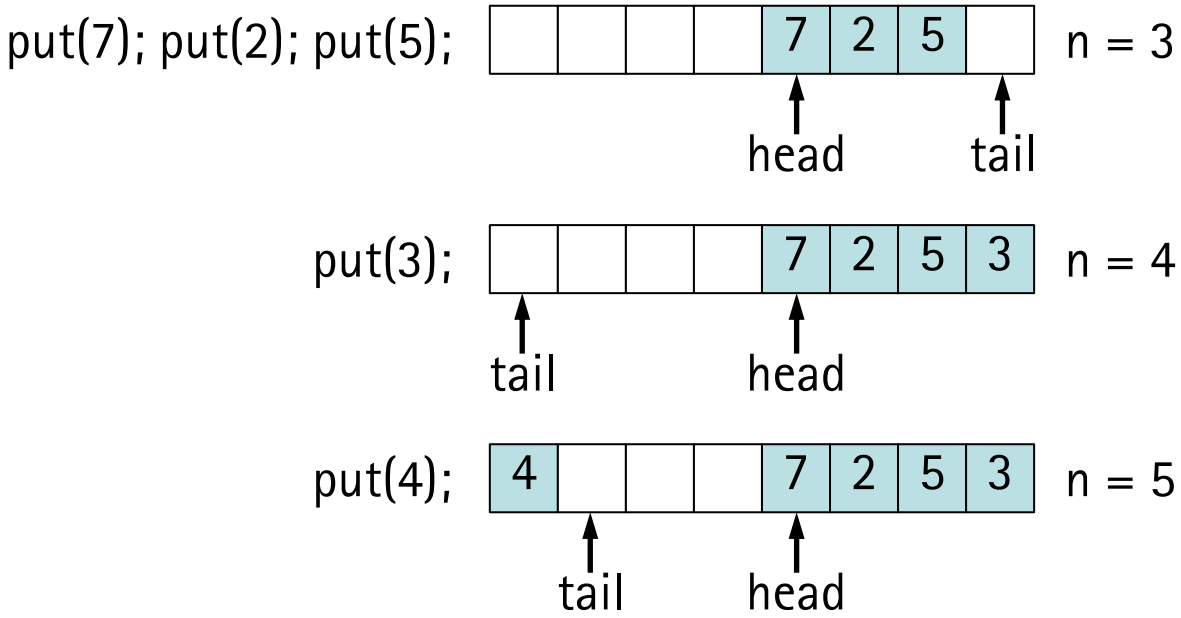
fixed size:
ring buffer

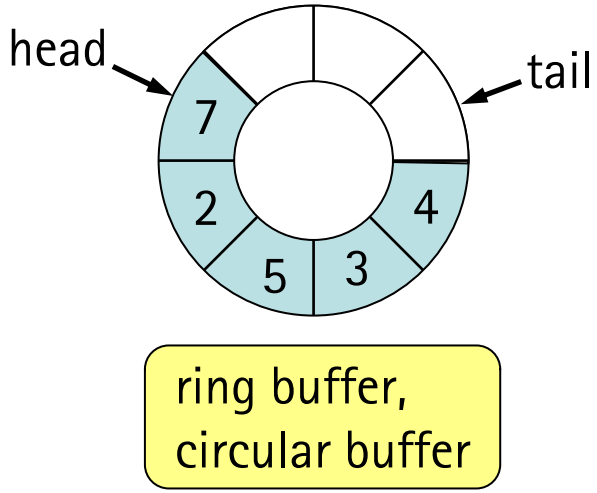
head: next pos. to read

tail: next position to write

Fixed Size Queue

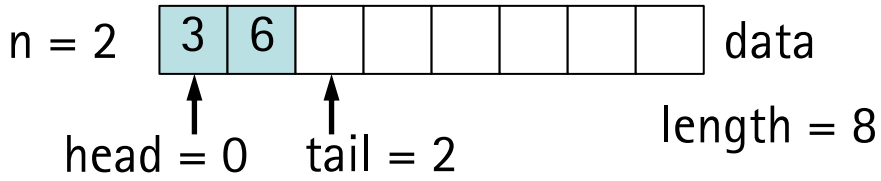
- head and tail are incremented modulo the size of the data array
  - head = (head+1) % data.length;
  - tail = (tail+1) % data.length;





Fixed Size Queue

```
typedef struct {
    int* data;
    int length; // capacity of the data array (number of ints)
    int head; // next position to read
    int tail; // next position to write
    int n; // number of items in the queue
} Queue;
```

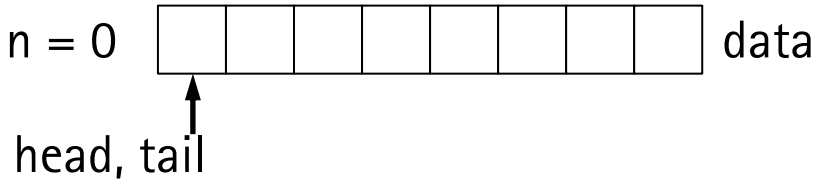


Fixed Size Queue

```
Queue* new_queue(int length) {
    require("positive", length > 0);
    Queue* q = xcalloc(1, sizeof(Queue));
    q->data = xcalloc(length, sizeof(int));
    q->length = length;
    q->head = 0;
    q->tail = 0;
    q->n = 0;
    return q;
}
```

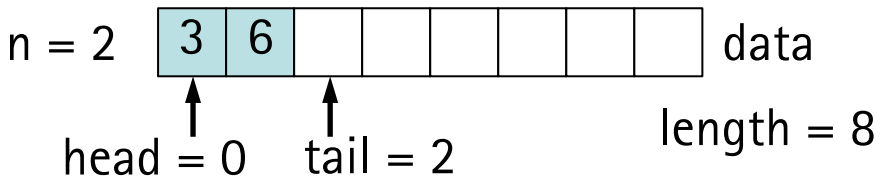
```
void free_queue(Queue* q) {
    require_not_null(q);
    free(q->data);
    free(q);
}
```

initial state:



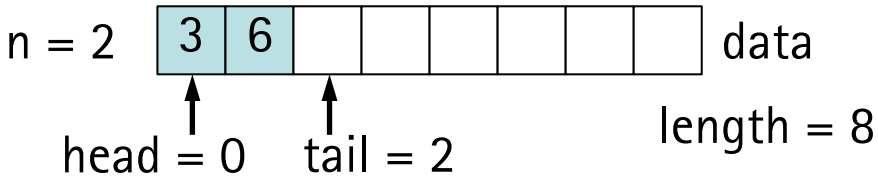
Fixed Size Queue

```
void put_queue(Queue* q, int x) {
    require_not_null(q);
    require("not full", q->n < q->length);
    q->data[q->tail] = x;
    q->n++;
    q->tail = (q->tail + 1) % q->length;
}
```



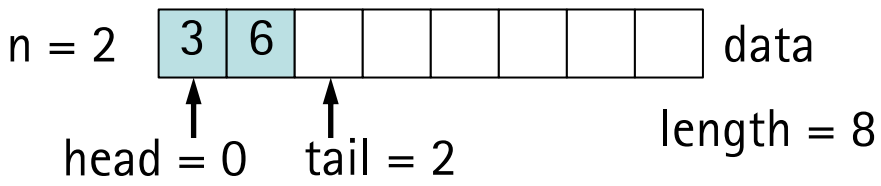
Fixed Size Queue

```
int get_queue(Queue* q) {
    require_not_null(q);
    require("not empty", q->n > 0);
    int x = q->data[q->head];
    q->n--;
    q->head = (q->head + 1) % q->length;
    return x;
}
```



Fixed Size Queue

```
int main(void) {
    report_memory_leaks(true);

    Queue* q = new_queue(8);
    put_queue(q, 3);
    put_queue(q, 6);
    println(get_queue(q)); // 3
    println(get_queue(q)); // 6

    free_queue(q);

    return 0;
}
```



STACK FOR PARSING POSTFIX EXPRESSIONS

Notation of Expressions: Infix, Prefix, Postfix

- Infix notation: Operator between operands
  - 1 + 2 * 3
  - Requires operator precedence rules and parentheses!
- Prefix notation: Operator before operands
  - + 1 * 2 3
  - evaluation right-to-left: + 1 * 2 3 → + 1 6 → 7
  - also called Polish Notation, no precedence rules
- Postfix notation: Operator after operands
  - 1 2 3 * +
  - evaluation left-to-right: 1 2 3 * + → 1 6 + → 7
  - also called Reverse Polish Notation
  - in postfix notation, operators appear when applied, no precedence rules

We only consider binary operators here (2 operands)

rm test.c test.o

Die Eier mit dem Zucker und dem Mehl mixen.

Evaluation of Expressions in Postfix Notation

- Using a stack to store intermediate results
- Example: 15 7 1 1 + - / 2 * infix: 15 / (7 - (1 + 1)) * 2

Token	Type	Stack	Description
15	operand	15	push operand onto stack
7	operand	15 7	push operand onto stack
1	operand	15 7 1	push operand onto stack
1	operand	15 7 1 1	push operand onto stack
+	operator	15 7 2	apply operator, push result
-	operator	15 5	apply operator, push result
/	operator	3	apply operator, push result
2	operand	3 2	push operand onto stack
*	operator	6	apply operator, push result

A Stack of Doubles

```
#define STACK_SIZE 100 // stack implemented as array, fixed capacity
typedef struct {
    double values[STACK_SIZE];
    int index; // next free stack entry
} Stack;

Stack stack1; // external variable, file level

// Removes all values from the stack.
void clear(Stack* s) {
    require_not_null(s);
    s->index = 0;
}
```

A Stack of Doubles

```
// Pushes a value on top of the stack.
void push(Stack* s, double f) {
    require_not_null(s);
    require("not full", s->index < STACK_SIZE);
    s->values[s->index++] = f;
}
```

equivalent to:

```
s->values[s->index] = f;
s->index++;
```

A Stack of Doubles

```
// Removes and returns value on top of the stack.
double pop(Stack* s) {
    require_not_null(s);
    require("not empty", s->index > 0);
    return s->values[--s->index];
}

// Checks whether the stack is empty.
bool is_empty(Stack* s) {
    require_not_null(s);
    return s->index <= 0;
}
```

equivalent to:

```
s->index--;
return s->values[s->index];
```

Using the Stack of Doubles

```
int main(void) {
    Stack* s = &stack1;
    clear(s);
    push(s, 3.5);
    push(s, 2.5);
    printdln(pop(s)); // output: 2.5
    printbln(is_empty(s)); // output: false
    printdln(pop(s)); // output: 3.5
    printbln(is_empty(s)); // output: true
    return 0;
}
```

Testing the Stack of Doubles

```
void stack_test(void) {
    Stack* s = &stack1;
    clear(s);
    push(s, 3.5);
    push(s, 2.5);
    test_within_d(pop(s), 2.5, 1e-6);
    test_equal_b(is_empty(s), false);
    test_within_d(pop(s), 3.5, 1e-6);
    test_equal_b(is_empty(s), true);
}

int main(void) {
    stack_test();
    return 0;
}
```

Output:

```
stack.c, line 19: check passed
stack.c, line 20: check passed
stack.c, line 21: check passed
stack.c, line 22: check passed
All 4 tests passed!
```

Postfix Expression Parser

```
double evaluate(Stack* s, char* expression) {
    return 0.0; // stub
}

void evaluate_test(Stack* s) {
    test_within_d(evaluate(s, "1.5 2.5 +"), 4.0, 1e-6);
    test_within_d(evaluate(s, "1.5 2.5 -"), -1.0, 1e-6);
    test_within_d(evaluate(s, " 100 0.01 * "), 1.0, 1e-6);
    test_within_d(evaluate(s, " 15  7  1  1  + - / 2 * "), 6.0, 1e-6);
}

int main(void) {
    evaluate_test(&stack1);
    return 0;
}
```

From Input String to Tokens

- Input is a string:
  - Example: `"1.5 2.5 + 123 *"` (15 characters, 5 tokens)
  - $token_1$: `"1.5"`, operand
  - $token_2$: `"2.5"`, operand
  - $token_3$: `"+"`, operator
  - $token_4$: `"123"`, operand
  - $token_5$: `"*"`, operator
- Split string into tokens: $token_1$, $token_2$, $token_3$ ...
- Identify token type: (floating-point) operand or (single character) operator

Representing Tokens

- Tokens are the relevant pieces of the input string

```
typedef struct {
    TokenType type;
    char* start; // inclusive
    char* end; // exclusive
} Token;
```

```
typedef enum {
    UNKNOWN,
    OPERAND,
    OPERATOR,
    END
} TokenType;
```

```
Token make_token(TokenType type, char* start, char* end) {
    require_not_null(start); require_not_null(end);
    Token t = { type, start, end };
    return t;
}
```

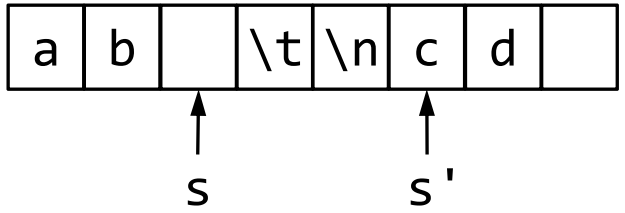
- How to split the input string into these pieces?

Helper Functions for Parsing

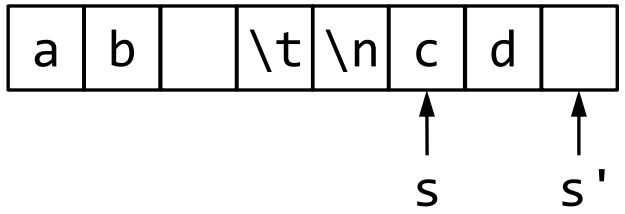
```
// May c appear in an operator?
bool is_operator_character(char c) {
    return c == '+' || c == '-' || c == '*' || c == '/';
}

// May c appear in a floating point number?
bool is_operand_character(char c) {
    return c == '.' || c == '-' || (c >= '0' && c <= '9');
}

// Is c whitespace?
bool is_whitespace(char c) {
   return c == ' ' || c == '\t' || c == '\r' || c == '\n';
}
```

```
// End of string reached?
bool eos(char* s) {
   return *s == '\0';
}
```

Helper Functions for Parsing




```
// Skips whitespace and returns a pointer to next
// non-whitespace character (or end of string).
char* skip_whitespace(char* s) {
    while (!eos(s) && is_whitespace(*s)) s++;
    assert("end of string reached or on first char of token", eos(s) || !is_whitespace(*s));
    return s;
}
```




```
// Skips token (non-whitespace) and returns pointer
// to next whitespace character (or to end of string).
char* skip_token(char* s) {
    while (!eos(s) && !is_whitespace(*s)) s++;
    assert("end of string reached or on first whitespace char after token", eos(s) || is_whitespace(*s));
    return s;
}
```

Helper Functions for Parsing

```
// Checks whether the chars from s (inclusive) to t (exclusive) are an operator.
bool is_operator(char* s, char* t) {
    return is_operator_character(*s) && t == s + 1;
}

// Checks whether the chars from s (inclusive) to t (exclusive) are an operand.
bool is_operand(char* s, char* t) {
    while (s < t) {
        if (!is_operand_character(*s)) return false;
        s++;
    }
    return true;
}
```

next_token

```
// Returns the next token.
Token next_token(char* s) {
    s = skip_whitespace(s);
    if (eos(s)) return make_token(END, NULL, NULL);
    assert("before end of string and on first char of token", !eos(s) && !is_whitespace(*s));
    char* t = skip_token(s);
    assert("end of string reached or on first whitespace char after token", eos(t) || is_whitespace(*t));
    if (is_operator(s, t)) {
        return make_token(OPERATOR, s, t);
    }
    if (is_operand(s, t)) {
        return make_token(OPERAND, s, t);
    }
    return make_token(UNKNOWN, s, t);
}
```

Postfix Expression Parser: Evaluate String

```
// Evaluates the given postfix expression.
double evaluate(Stack *stack, char* expression) {
    double operand, operand1, operand2; char operator;
    while (true) {
        Token token = next_token(expression);
        switch (token.type) {
            case OPERAND: operand = atof(token.start); // atof: ascii-to-float, stdlib.h
                          push(stack, operand);
                          break;
            case OPERATOR: operator = token.start[0]; // we only have single-character, binary operators
                           operand2 = pop(stack);
                           operand1 = pop(stack);
                           push(stack, apply(operator, operand1, operand2));
                           break;
            case END: return pop(stack);
            case UNKNOWN: print_token(token); exit(1);
        }
        expression = token.end; // continue after the current token
    }
}
```

Postfix Expression Parser: Apply Operator

```
// Applies the operator to the operands.
double apply(char operator, double operand1, double operand2) {
    switch (operator) {
        case '+': return operand1 + operand2;
        case '-': return operand1 - operand2;
        case '*': return operand1 * operand2;
        case '/': return operand1 / operand2;
        default: {
            printf("Unknown operator: %c\n", operator);
            exit(1);
        }
    }
    return 0;
}
```

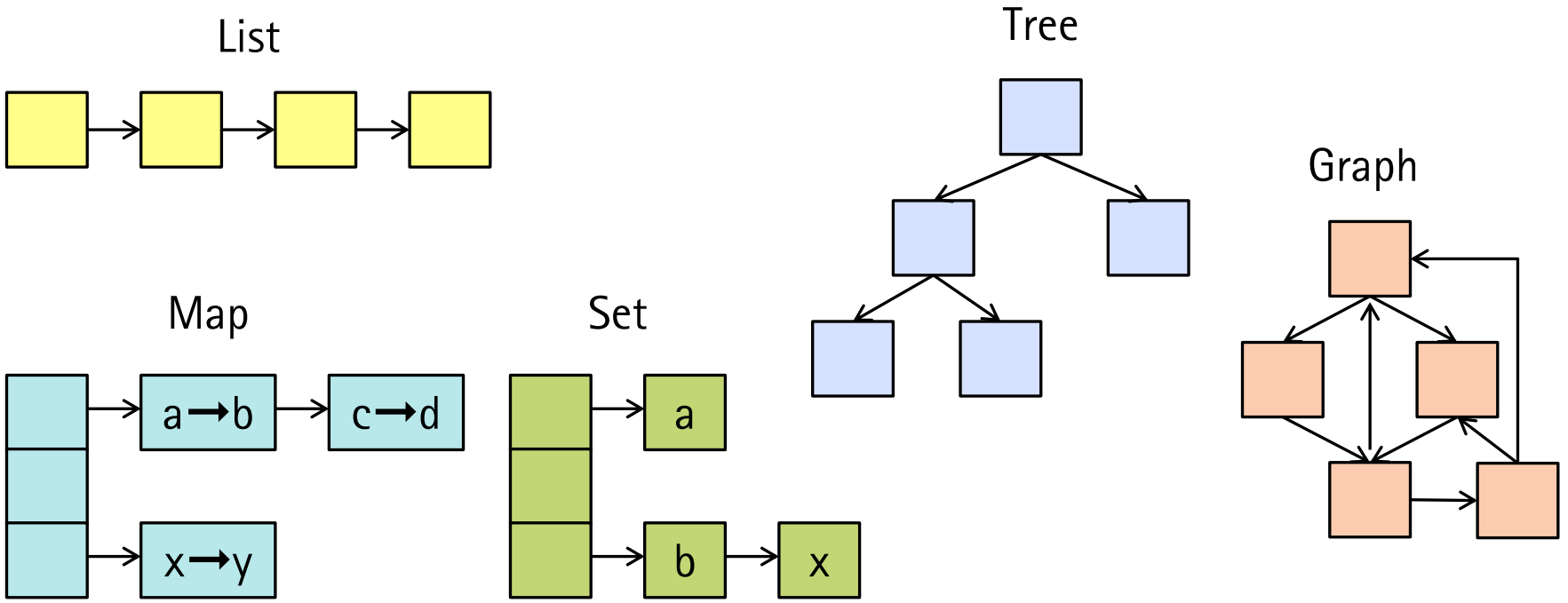
Postfix Expression Parser: Command Line Input

```
int main(void) {
    while (true) {
        // read line from standard input (at most 100 characters)
        char* expression = s_input(100);
        // finish if input contains “exit”
        if (s_contains(expression, "exit")) break;
         // evaluate postfix expression using stack
        printdln(evaluate(&stack1, expression));
    }
    return 0;
}
```

DYNAMIC DATA STRUCTURES

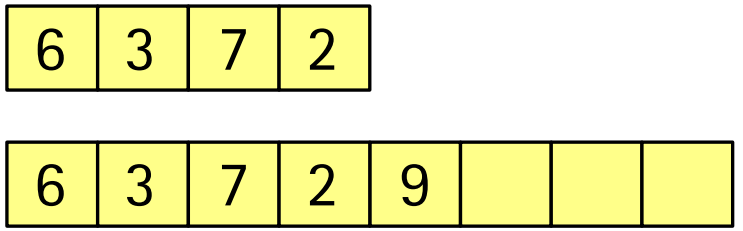
Dynamic Data Structures

- Elements and nodes are dynamically created at runtime
- Data structure can grow and shrink as needed
- Important dynamic data structures:



Vector (Dynamic Array)

- Linear sequence, grows as needed, access by index
- Elements are stored in a C array
- If capacity is exhausted, a larger array is allocated and all elements are copied to the new array
  - Typically $size_{new} = 2 * size_{old}$



- Access to arbitrary elements by index is very efficient
  - Computation of address of array element
- Inserting between elements is expensive
  - Need to shift up all elements above insertion position
- Inserting at the end is efficient (if capacity is not exhausted)

Vector (Dynamic Array)

```
typedef struct {
    int capacity; // maximum number of items
    int count; // current number of items
    int* data;
} Vector;

Vector* vector_new(int capacity) {
    require("positive", capacity > 0);
    Vector* v = xcalloc(1, sizeof(Vector));
    v->capacity= capacity; // initial capacity
    v->data = xcalloc(v->capacity, sizeof(int));
    v->count = 0;
    return v;
}
```

```
void vector_free (Vector* v) {
    require_not_null(v);
    free(v->data);
    free(v);
}
```

Vector (Dynamic Array)

```
void vector_add(Vector* v, int x) {
    require_not_null(v);
    if (v->count >= v->capacity) { // if full, reallocate
        v->capacity *= 2; // duplicate capacity
        fprintf(stderr, "reallocating, new capacity: %d ints\n", v->capacity);
        int* data_new = xcalloc(v->capacity, sizeof(int)); // allocate larger array
        memcpy(data_new, v->data, v->count * sizeof(int)); // copy elements
        free(v->data); // delete old array
        v->data = data_new; // set new array
    }
    v->data[v->count++] = x; // store value
}
```

```
#include <string.h>
void* memcpy(void* dst, const void* src, size_t n)
typedef long unsigned int size_t;
```

Vector (Dynamic Array)

```
int vector_count(Vector* v) {
    require_not_null(v);
    return v->count;
}

int vector_get(Vector* v, int i) {
    require_not_null(v);
    require("valid index", i >= 0 && i < v->count);
    return v->data[i];
}
```

Vector (Dynamic Array)

```
int main(void) {
    report_memory_leaks(true);
    Vector* v = vector_new(2);
    for (int i = 0; i < 100; i++) {
        vector_add(v, i_rnd(100)); // add random numbers in range 0..99
    }
    printf("count = %d\n", vector_count(v));
    for (int i = 0; i < 100; i++) {
        printf("%d, ", vector_get(v, i));
    }
    printf("\n");
    vector_free(v);
    return 0;
}
```

Vector (Dynamic Array)

```
$ ./vector
reallocating, new capacity: 4 ints
reallocating, new capacity: 8 ints
reallocating, new capacity: 16 ints
reallocating, new capacity: 32 ints
reallocating, new capacity: 64 ints
reallocating, new capacity: 128 ints
count = 100
92, 75, 35, 96, 32, 20, 61, 56, 37, 26, 82, 85, 70, 9, 56, 9, 52, 71, 19, 52,
24, 44, 20, 42, 72, 16, 33, 29, 4, 78, 16, 34, 20, 53, 71, 48, 83, 28, 84, 11,
61, 59, 17, 74, 89, 64, 65, 65, 43, 94, 27, 83, 11, 24, 82, 98, 13, 60, 83, 0,
30, 82, 25, 14, 31, 92, 86, 10, 45, 79, 15, 39, 41, 75, 43, 74, 17, 24, 25, 70,
8, 64, 65, 32, 23, 69, 86, 75, 88, 35, 66, 46, 9, 31, 28, 74, 81, 64, 60, 70,
```

initial size 2,
n = 100 elements,
r = 6 reallocations
= $\lceil \log_2(n) \rceil$

Vector (Dynamic Array)

```
typedef struct {
    int capacity; // maximum number of items
    int count; // current number of items
    int* data;
} Vector;
```

```
Vector* vector_new(int capacity) {
    require("positive", capacity > 0);
    Vector* v = xcalloc(1, sizeof(Vector));
    v->capacity= capacity; // initial capacity
    v->data = xcalloc(v->capacity, sizeof(int));
    v->count = 0;
    return v;
}
```

```
void vector_free (Vector* v) {
    require_not_null(v);
    free(v->data);
    free(v);
}
```

Vector with Flexible Array

```
typedef struct {
    int capacity;  // maximum number of items
    int count;     // current number of items
    int data[];    // flexible array member must be last
} Vector;
```

```
Vector* vector_new(int capacity) {
    require("positive", capacity > 0);
    int n = sizeof(Vector) + capacity * sizeof(int);
    Vector* v = xmalloc(n);
    memset(v, 0, n); // set each byte to 0
    v->capacity = capacity;
    return v;
}
```

```
void vector_free (Vector* v) {
    require_not_null(v);
    free(v);
}
```

Vector with Flexible Array: add

```
Vector* vector_add(Vector* v, int x) {
    require_not_null(v);
    if (v->count >= v->capacity) { // if full, reallocate
        Vector* vnew = vector_new(2 * v->capacity); // duplicate capacity
        fprintf(stderr, "reallocating, new capacity: %d ints\n", vnew->capacity);
        memcpy(vnew->data, v->data, v->count * sizeof(int)); // copy elements
        vnew->count = v->count;
        free(v); // delete old vector
        v = vnew; // set new array
    }
    v->data[v->count++] = x; // store value
    return v;
}
```

has to return potentially reallocated vector

Dynamic Stack

- Compared to array-stack
  - Stack grows as needed
  - Implementation changed (slightly less efficient)
  - Interface almost unchanged (no size parameter in constructor)

- Information hiding
  - Internal structure is hidden from clients
  - Implementation can change without having to change clients

```
typedef struct {
    Node* top;
} Stack;

Stack* new_stack(void) {
    Stack* s = xcalloc(1, sizeof(Stack));
    s->top = NULL;
    return s;
}
```

no size parameter

Dynamic Stack

```c
typedef struct Node {
    int value;
    struct Node* next;
} Node;

Node* new_node(int value, Node* next) {
    Node* n = xcalloc(1, sizeof(Node));
    n->value = value;
    n->next = next;
    return n;
}
```

```c
typedef struct {
    Node* top;
} Stack;

Stack* new_stack(void) {
    Stack* s = xcalloc(1, sizeof(Stack));
    s->top = NULL;
    return s;
}
```

Dynamic Stack

```
void free_stack(Stack* s) {
    require_not_null(s);
    Node* next;
    for (Node* n = s->top; n != NULL; n = next) {
        next = n->next;
        free(n);
    }
    free(s);
}
```

```
void push(Stack* s, int x) {
    require_not_null(s);
    s->top = new_node(x, s->top);
}
```

```
bool empty_stack(Stack* s) {
    require_not_null(s);
    return s->top == NULL;
}
```

Dynamic Stack

```
int pop(Stack* s) {
    require_not_null(s);
    require("not empty", s->top != NULL);
    Node* t = s->top;
    int x = t->value;
    s->top = t->next;
    free(t);
    return x;
}
```

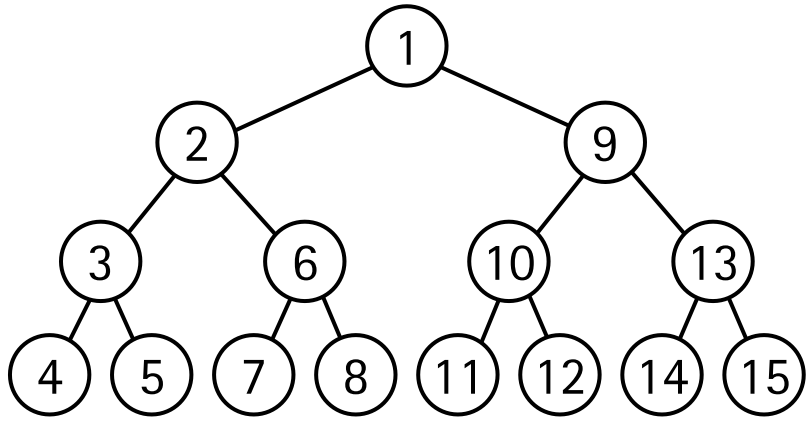
```
int main(void) {
    base_init();
    base_set_memory_check(true);

    Stack* s = new_stack();
    push(s, 3);
    push(s, 6);
    println(pop(s)); // 6
    println(pop(s)); // 3
    printbln(empty_stack(s)); // true

    free_stack(s);
    return 0;
}
```

Iteratively Traversing a Binary Tree with a Stack

- Up to now have used recursive algorithms to process trees
- May use iterative algorithms (without recursion) to process trees
- Need a way to store tree nodes that still have to be processed
- For binary tree: In each step can only take one path (e.g., left), have to remember other child (e.g., right) for later processing

Iteratively Traversing a Binary Tree with a Stack

```
int main(void) {
    BTNode* t = new_btnode(1,
                        new_btnode(2,
                            new_btnode(3, new_leaf(4),  new_leaf(5)),
                            new_btnode(6, new_leaf(7), new_leaf(8))),
                        new_btnode(9,
                            new_btnode(10, new_leaf(11), new_leaf(12)),
                            new_btnode(13, new_leaf(14), new_leaf(15))));
    println(sum(t));
    free_tree(t);
    return 0;
}
```



Iteratively Traversing a Binary Tree with a Stack

```
int sum(BTNode* tree) {
    int sum = 0;
    Stack* stack = new_stack(); // for storing nodes that still have to be processed
    push(stack, tree); // push root node
    while (!empty_stack(stack)) { // while there are more nodes to process
        BTNode* n = pop(stack); // take a node
        sum += n->value; // process it (add its value to the sum)
        if (n->right != NULL) push(stack, n->right); // add right child to stack
        if (n->left != NULL) push(stack, n->left); // add left child to nodes stack
    }
    // assert: stack is empty, all nodes have been processed
    free_stack(stack);
    return sum;
}
```

what would happen when swapping these if-statements?

Iteratively Traversing a Binary Tree with a Stack

```
typedef struct PointerNode {
    void* value;
    struct PointerNode* next;
} PointerNode;


void push(Stack* s, void* x) {
    require_not_null(s);
    s->top = new_pointer_node(x, s->top);
}
```

```
void* pop(Stack* s) {
    require_not_null(s);
    require("not empty", s->top != NULL);
    PointerNode* t = s->top;
    void* x = t->value;
    s->top = t->next;
    free(t);
    return x;
}
```

Dynamic Queue

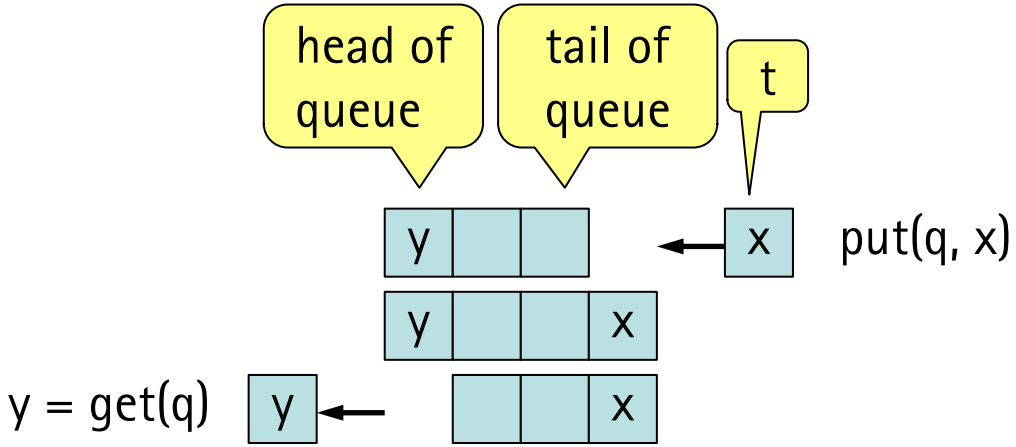
```
typedef struct {
    Node* head; // front of queue (first to be served)
    Node* tail; // back of queue (last to be served)
} Queue;

Queue* new_queue(void) {
    Queue* q = xcalloc(1, sizeof(Queue));
    q->head = NULL; // not necessary if using xcalloc
    q->tail = NULL; // not necessary if using xcalloc
    return q;
}
```

Dynamic Queue

```
void free_queue(Queue* q) {
    require_not_null(q);
    Node* next;
    for (Node* n = q->head; n != NULL; n = next) {
        next = n->next;
        free(n);
    }
    free(q);
}

bool empty_queue(Queue* q) {
    require_not_null(q);
    return q->head == NULL;
}
```

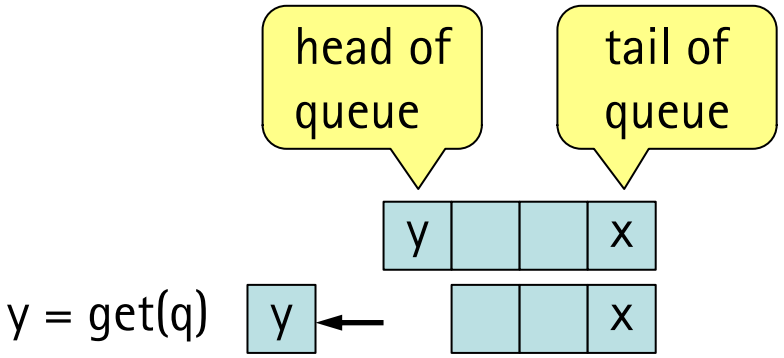
Dynamic Queue

```
void put_queue(Queue* q, int x) {
    require_not_null(q);
    Node* t = new_node(x, NULL);
    if (q->tail != NULL) q->tail->next = t;
    q->tail = t;
    if (q->head == NULL) q->head = t;
}
```



Dynamic Queue

```
int get_queue(Queue* q) {
    require_not_null(q);
    require("not empty", q->head != NULL);
    Node* h = q->head;
    int y = h->value;
    q->head = h->next;
    if (q->head == NULL) q->tail = NULL;
    free(h);
    return y;
}
```



Dynamic Queue

```
int main(void) {
    report_memory_leaks(true);

    Queue* q = new_queue();
    put_queue(q, 3);
    put_queue(q, 6);
    while (!empty_queue(q)) {
        printf("%d ", get_queue(q)); // 3 6
    }
    printf("\n");
    free_queue(q);
    return 0;
}
```

Dynamic Queue

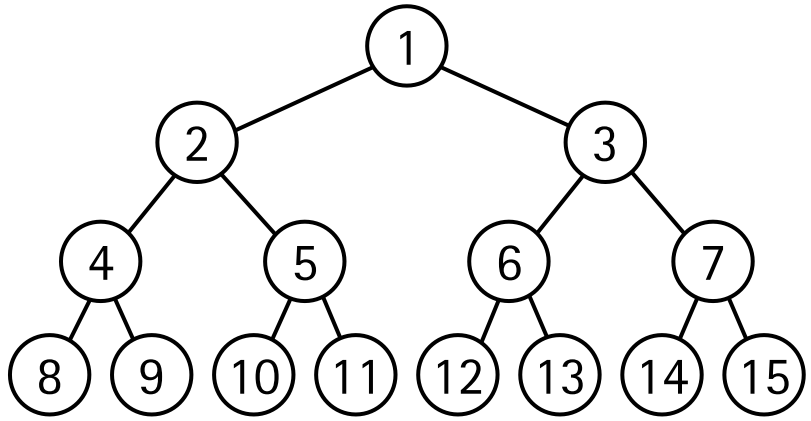
- Compared to array-queue
  - Queue grows as needed
  - Slightly less efficient than fixed-size queue
  - Implementation changed, interface almost unchanged
    - (except: no size parameter in constructor)
- Information hiding
  - Internal structure is hidden from clients
  - Implementation can change without having to change clients

Iteratively Traversing a Binary Tree with a Queue

- Up to now have used recursive algorithms to process trees
- May use iterative algorithms (without recursion) to process trees
- Need a way to store tree nodes that still have to be processed
- For binary tree: In each step can only take one path (e.g., left), have to remember other node (e.g., right) for future processing

Iteratively Traversing a Binary Tree with a Queue

```
int main(void) {
    BTNode* t = new_btnode(1,
                        new_btnode(2,
                            new_btnode(4, new_leaf(8),  new_leaf(9)),
                            new_btnode(5, new_leaf(10), new_leaf(11))),
                        new_btnode(3,
                            new_btnode(6, new_leaf(12), new_leaf(13)),
                            new_btnode(7, new_leaf(14), new_leaf(15))));
    println(sum(t));
    free_tree(t);
    return 0;
}
```



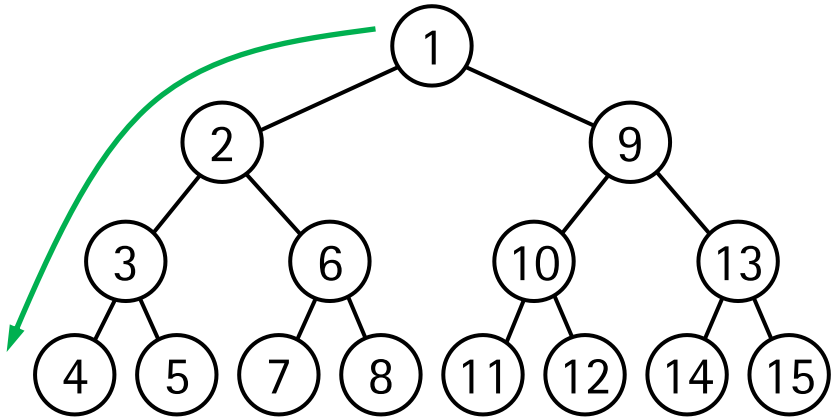
Iteratively Traversing a Binary Tree with a Queue

```
int sum(BTNode* tree) {
    int sum = 0;
    Queue* q = new_queue(); // for storing nodes that still have to be processed
    put_queue(q, tree); // enqueue root node
    while (!empty_queue(q)) { // while there are more nodes to process
        BTNode* n = get_queue(q); // take one of them
        sum += n->value; // process node (add its value to the sum)
        if (n->left != NULL) put_queue(q, n->left); // add left child to queue
        if (n->right != NULL) put_queue(q, n->right); // add right to queue
    }
    // assert: queue is empty, all nodes have been processed
    free_queue(q);
    return sum;
}
```

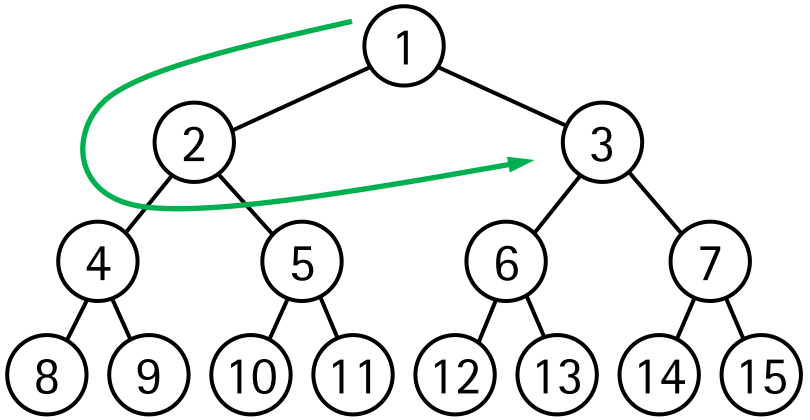
what would happen when swapping these if-statements?

Stack Traversal vs. Queue Traversal

- Stack: Depth first
  - Vertical first, then horizontal



- Queue: Breadth first
  - Horizontal first, then vertical



Map

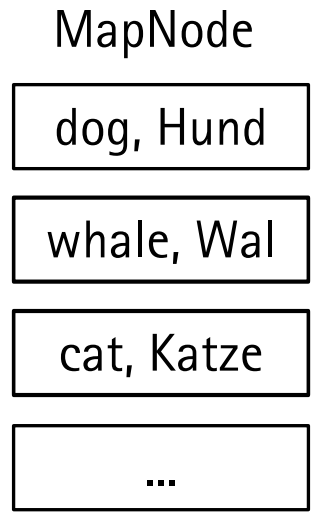
- Associative array with efficient lookup
  - Associate “keys” to values
  - Store a value: put_map(map, key, value)
  - Lookup a value: value = get_map(map, key)
  - Arrays map integer indices to arbitrary values
  - Maps map arbitrary values to arbitrary values
- Example: English-to-German dictionary is a Map of strings to strings
  - Dictionary contains key ⟶ value associations:
  - dog ⟶ Hund, cat ⟶ Katze, car ⟶ Auto
  - translation = get_map(dictionary, "dog")

Mapping (key-value pair, Association)

- Element of a map associates a key to a value
  - Also called: key-value pair

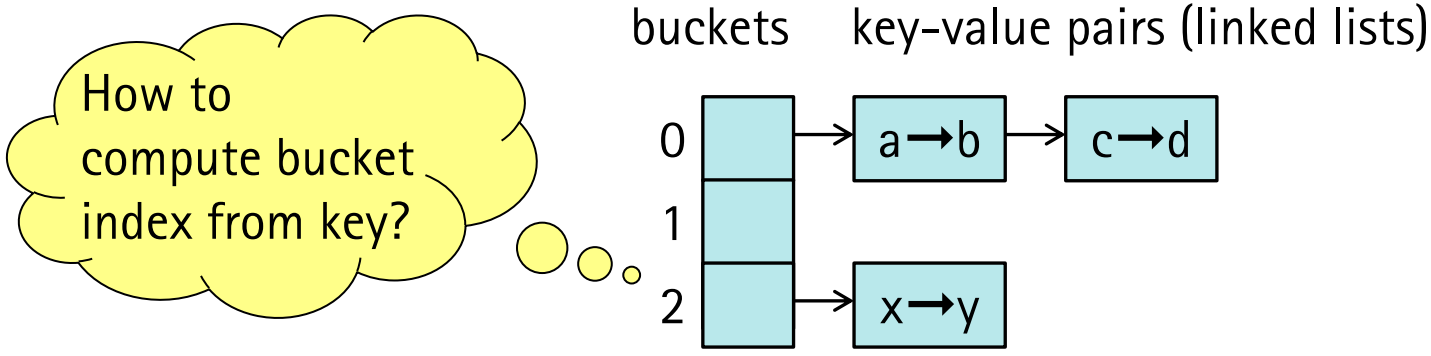
```
typedef struct Entry {
    void* key; // e.g., dog
    void* value; // e.g., Hund
} Entry;
```

- Given a key (and many key-value pairs), efficiently find the value



Map implemented as a Hash Table

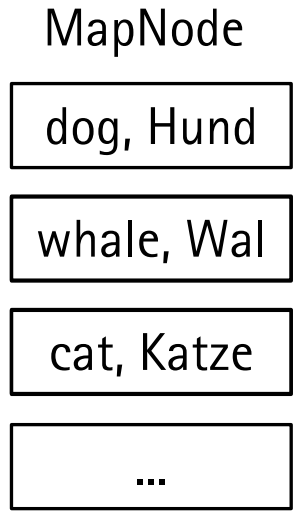
- Hash table: Concrete implementation of abstract data type map
- Hash table has an array of "buckets"
  - Each bucket stores a list of key⟶value associations
  - Keys are mapped to integer hash values: int h = hash(key)
  - Bucket index is computed from hash value: b = buckets[h % buckets.length]



Mappings (key-value pairs, Associations)

- Need linked lists of key-value pairs

```
typedef struct MapNode {
    Entry entry; // not a pointer, but inline
    struct MapNode* next; // next in the list with that hash
} MapNode;

MapNode* new_map_node(void* key, void* value, MapNode* next) {
    require_not_null(key); require_not_null(value);
    MapNode* node = xcalloc(1, sizeof(MapNode));
    node->entry.key = key;
    node->entry.value = value;
    node->next = next;
    return node;
}
```



Example Hash Function for Strings

- Hash function

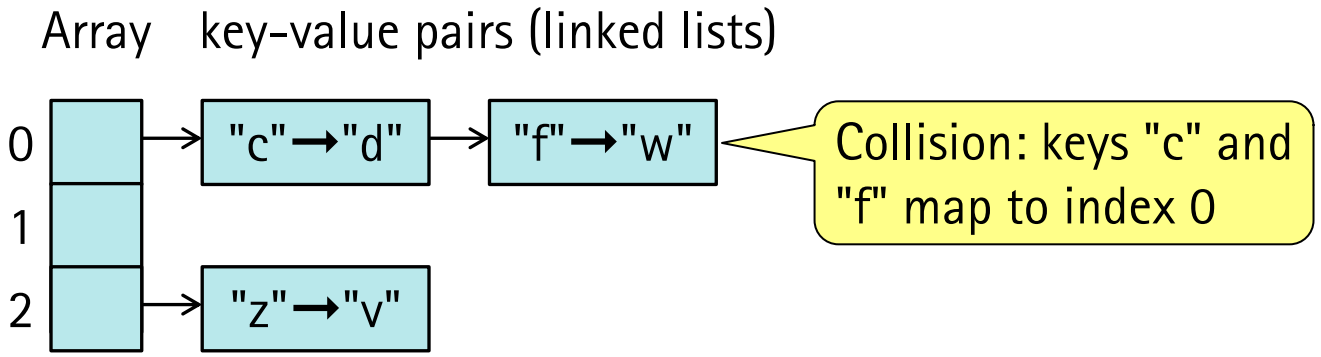
```
typedef int (*HashFunc)(void*);
```

```
int hash_string(void* x) {
    String key = x;
    int n = s_length(key);
    int hash = 0;
    for (int i = 0; i < n; i++) { hash = hash * 31 + key[i]; }
    if (hash < 0) hash = -hash;
    return hash;
}
```

- Example: Collision for “dog” and “donkey” (for 4 buckets)
  - hash("dog") = 99644, hash("dog") % 4 = 0
  - hash("donkey") = 1326158276, hash("donkey") % 4 = 0
  - hash("cow") = 98699, hash("cow") % 4 = 3

Good Hash Functions?

- Computes array index (type int) from key (e.g., type String)
- Should be efficient
- Should distribute keys evenly over available buckets
  - If two keys map to the same bucket index, then there is a collision
  - This typically cannot be avoided, so has to be resolved afterwards
  - The longer the list in each bucket, the less efficient the hash table
  - May need to reallocate a larger bucket array if filled too much...



Map: Create a Map

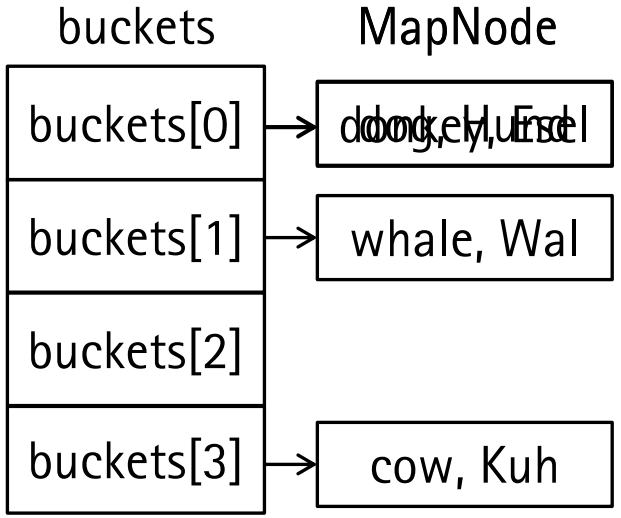
```
typedef struct {
    int n_buckets; // number of buckets
    MapNode** buckets; // array of buckets (each elem. is a ptr. to a key-value pair)
    HashFunc hash; // the hash function for the keys
    EqualFunc equal;
} Map;

Map* new_map(int n_buckets, HashFunc hash, EqualFunc equal) {
    Map* m = xcalloc(1, sizeof(Map));
    m->n_buckets = n_buckets;
    m->buckets = xcalloc(n_buckets, sizeof(MapNode*));
    m->hash = hash;
    m->equal = equal;
    return m;
}
```

```
typedef bool (*EqualFunc)(void* x, void* y);
```

Map: Add a Key-Value Pair to the Map

```
void put_map(Map* m, void* key, void* value) {
    int h = m->hash(key) % m->n_buckets;
    // add key-value association to front of list in bucket h
    // (does not check whether key is already in map!)
    m->buckets[h] = new_map_node(key, value, m->buckets[h]);
}
```

```
put("dog", "Hund"); // 0
put("whale", "Wal"); // 1
put("cow", "Kuh"); // 3
put("donkey", "Esel"); // 0
```



Map: Look Up the Value Associated with a Key

```
void* get_map(Map* m, void* key) {
    int h = m->hash(key) % m->n_buckets;
    // walk through the list of bucket h
    for (MapNode* node = m->buckets[h]; node != NULL; node = node->next) {
        if (m->equal(key, node->entry.key)) { // use equals method to compare keys
            return node->entry.value; // found key, return associated value
        }
    }
    return NULL; // key not found, return null
}
```

Map: Free the Map

```
void free_map(Map* m, FreeFunc free_key, FreeFunc free_value) {
    MapNode* next;
    for (int i = 0; i < m->n_buckets; i++) {
        for (MapNode* node = m->buckets[i]; node != NULL; node = next) {
            next = node->next;
            if (free_key != NULL) free_key(node->entry.key);
            if (free_value != NULL) free_value(node->entry.value);
            free(node); // MapNode structure
        }
    }
    free(m->buckets); // buckets array
    free(m); // Map structure
}
```

```
typedef void (*FreeFunc)(void* object);
```

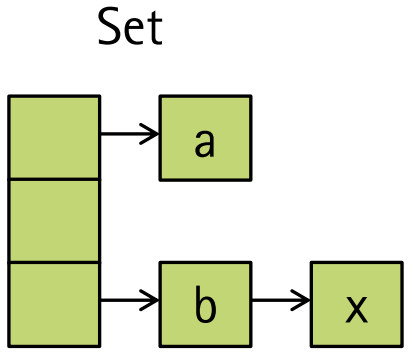
Map: Example Usage

```
int main(void) {
    base_init(); base_set_memory_check(true);
    Map* m = new_map(4, hash_string, strs_equal);
    put_map(m, "dog", "Hund");
    put_map(m, "cat", "Katze");
    put_map(m, "whale", "Wal");
    put_map(m, "cow", "Kuh");
    put_map(m, "donkey", "Esel");
    printsln(get_map(m, "cow")); // Kuh
    printsln(get_map(m, "dog")); // Hund
    printsln(get_map(m, "donkey")); // Esel
    printsln(get_map(m, "hamster")); // <null>, not in m
    free_map(m, free_string);
    return 0;
}
```

```
void free_string(void* s) {
    free(s);
}
bool strs_equal(void* x, void* y) {
    return s_equals(x, y);
}
```

Set

- Similar to map, but only stores keys, not values
  - No repeated elements
  - Elements not ordered
- Set elements stored in a hash table
- Typical set operations
  - add, remove – element addition and removal
  - contains – set membership
  - is_empty
  - size – number of elements
  - union, intersection, difference, is_subset_of
  - iterator – enumerate all elements of the set



Set Element

```
typedef struct SetNode {
    void* element;
    struct SetNode* next;
} SetNode;

SetNode* new_set_node(void* element, SetNode* next) {
    SetNode* node = xcalloc(1, sizeof(SetNode));
    node->element = element;
    node->next = next;
    return node;
}
```

Set: Create a Set

```
typedef struct {
    int n_buckets; // number of buckets
    SetNode** buckets; // array of buckets (each element is a pointer to an element)
    int n_elements; // number of elements currently in the set
    HashFunc hash; EqualFunc equal; // functions for hashing and equality checking
} Set;

Set* new_set(int n_buckets, HashFunc hash, EqualFunc equal) {
    Set* s = xcalloc(1, sizeof(Set));
    s->n_buckets = n_buckets;
    s->buckets = xcalloc(n_buckets, sizeof(SetNode*));
    s->n_elements = 0;
    s->hash = hash;    s->equal = equal;
    return s;
}
```

Set: Query a Set

```
bool contains_set(Set* s, void* key) {
    int h = s->hash(key) % s->n_buckets;
    // walk through the list of bucket h
    for (SetNode* node = s->buckets[h]; node != NULL; node = node->next) {
        if (s->equal(key, node->element)) { // use equals method to compare keys
            return true; // found key
        }
    }
    return false; // key not found
}
```

```
bool is_empty_set(Set* s) {
    return s->n_elements == 0;
}
```

```
int size_set(Set* s) {
    return s->n_elements;
}
```

Set: Add an Element

```
void add_set(Set* s, void* key) {
    if (!contains_set(s, key)) { // do not allow multiple membership
        int h = s->hash(key) % s->n_buckets;
        // add key-value association to front of list in bucket h
        // (does not check whether key is already in set!)
        s->buckets[h] = new_set_node(key, s->buckets[h]);
        s->n_elements++; // one more element
    }
}
```

Set: Free the Set

```
void free_set(Set* s, FreeFunc free_element) {
    SetNode* next;
    for (int i = 0; i < s->n_buckets; i++) {
        for (SetNode* node = s->buckets[i]; node != NULL; node = next) {
            next = node->next;
            if (free_element != NULL) free_element(node->element);
            free(node); // SetNode structure
        }
    }
    free(s->buckets); // buckets array
    free(s); // Set structure
}
```

Iterators: Abstracting Iteration

- Enumerating the elements of a collection without knowing the internal structure of the collection
- An iterator stores the current iteration state
  - Knows whether there are more objects
  - Returns objects one-by-one
- Using a set iterator:

```
Set* set = new_set(1024, hash_string, strings_equal);
// add string elements to the set...
Iterator* iter = new_iterator(set);
while (has_next(iter)) { // while there are more elements
    String element = next(iter); // get the next one
    printf("%s\n", element); // assume string elements
}
free_iterator(iter);
```

iterator abstracts from complicated iteration over sets

Iterators: Creation

- Iterators need to keep track of the iteration state

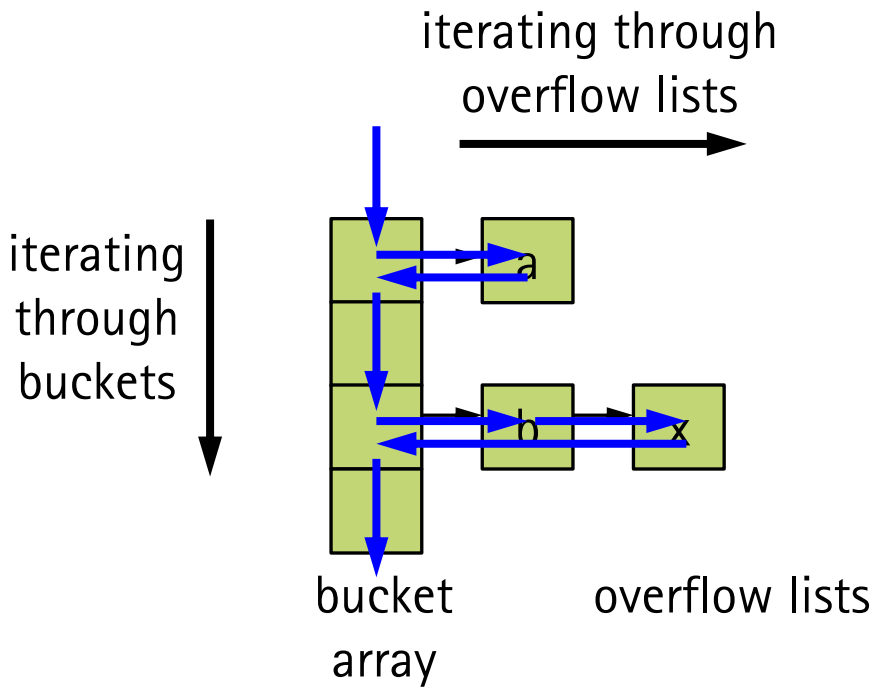
```
typedef struct {
    Set* s; // the set over which this iterator iterates
    int bucket_index; // current bucket list index
    SetNode* node; // current node in list
} Iterator;
```

- Creating an iterator

```
Iterator* new_iterator(Set* s) {
    Iterator* i = xcalloc(1, sizeof(Iterator));
    i->s = s;
    i->bucket_index = -1;
    i->node = NULL;
    return i;
}
```

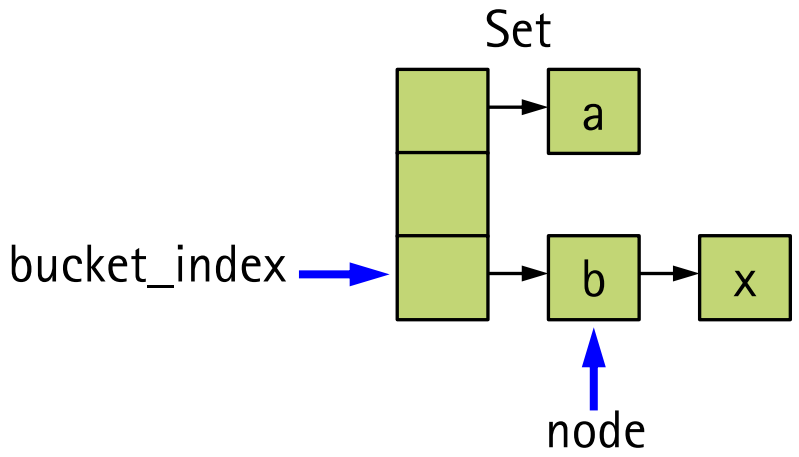
Iterators: How to Iterate through a Set

- No specific order of elements in a set, because
  - no order specified on the elements
  - elements are put in buckets based on hash value



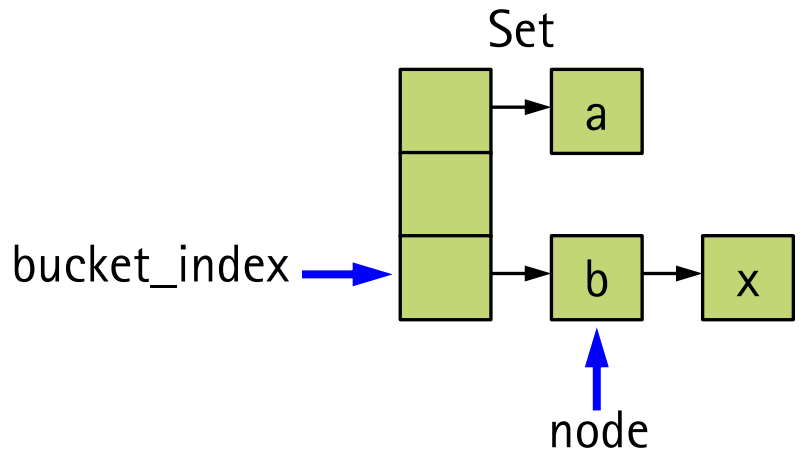
Iterators: Check if there are more elements

```
bool has_next(Iterator* iter) { // does not change state of iterator
    if (iter->node != NULL && iter->node ->next != NULL) { // in list?
        return true; // there is another element in this bucket list
    } else { // try next bucket lists
        Set* s = iter->s;
        for (int i = iter->bucket_index + 1; i < s->n_buckets; i++) {
            if (s->buckets[i] != NULL) return true; // bucket i is not empty
        }
    }
    return false; // no more elements
}
```



Iterators: Get the Next Element (if any)

```
void* next(Iterator* iter) { // modifies state of iterator
    if (iter->node != NULL && iter->node->next != NULL) {
        iter->node = iter->node->next; // advance to next elem.
        return iter->node->element;
    } else {
        Set* s = iter->s;
        for (int i = iter->bucket_index + 1; i < s->n_buckets; i++) {
            if (s->buckets[i] != NULL) {
                iter->bucket_index = i;
                iter->node = s->buckets[i]; // advance to next element
                return iter->node->element;
            }
        }
    }
    return NULL; // no more elements
}
```



Use Set Iterator to Compute Intersection

```
Set* intersection(Set* a, Set* b) {
    require("equal element types", a->hash == b->hash && a->equal == b->equal);
    Set* result = new_set(a->n_buckets, a->hash, a->equal);
    Iterator* iter = new_iterator(a);
    while (has_next(iter)) {
        void* e = next(iter); // get element e from set a
        if (contains_set(b, e)) add_set(result, e); // add to result if e is also in b
    }
    free_iterator(iter);
    return result;
}
```

Testing the Set

```
Set* s = new_set(4, hash_string, strings_equal);
printbln(is_empty_set(s)); // true
add_set(s, "dog");
printbln(is_empty_set(s)); // false
add_set(s, "cat");
add_set(s, "hamster");
add_set(s, "hamster"); // already in set, not added again
printiln(size_set(s)); // 3
printbln(contains_set(s, "hamster")); // true
printbln(contains_set(s, "donkey")); // false
```

Testing the Set

```
Set* t = new_set(4, hash_string, strings_equal);
add_set(t, "cow");
add_set(t, "cat");
add_set(t, "dog");
Set* r = intersection(s, t);    // { "dog", "cat", "hamster" } ∩ { "cow", "cat", "dog" }
iter = new_iterator(r);     // r = { "dog", "cat" }
while (has_next(iter)) {
    String element = next(iter);
    printsln(element);
}
free_iterator(iter);
free_set(s, NULL);
free_set(t, NULL);
free_set(r, NULL);
```

Summary

- Abstract data types concept
- List: Linear sequence of elements, dynamic
- Vector: A dynamic array
- Stack: LIFO, array-based or list-based (dynamic)
- Queue: FIFO, array-based or list-based (dynamic)
- Map: Associations of keys to values, implemented as hash tables
- Set: Unordered, non-repeating collection of elements
- Iterators abstract from going through complicated structures
- Tree: Hierarchy of elements, sorted trees for searching