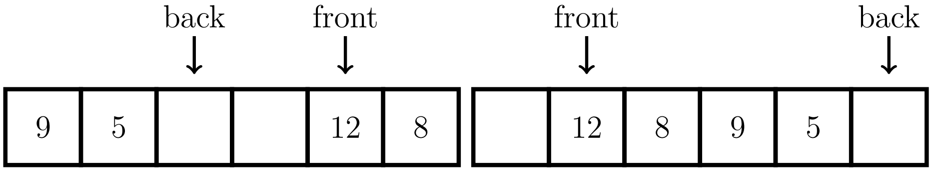
Aufgabe 1

Wir implementieren die Queue als zirkuläres dynamisches Array. Wir verwalten hierzu zwei Indizes `front` und `back`, welche auf das Element vor dem letzen bzw. auf das erste Element der Schlange verweisen (siehe Abbildung). Unsere Schlange wird immer einen kompletten Teil des Arrays belegen, wenn man es sich zirkulär vorstellt, d.h. der Nachfolger des letzten Elements ist das erste. Die Schlange $(2, 3, 4)$ könnte z.B. durch das Array $(4, , 2, 3)$ dargestellt werden, wobei der Front-Zeiger auf 3 und der Back-Zeiger auf 1 stehen würden (das Array sei hierbei 1-indiziert).

Abbildung 1: Dequeue $(12, 8, 9, 5)$ als zirkuläres Array, beide möglichen Fälle



Beim Einfügen vorne müssen wir testen, ob das Arrayelement hinter dem `front`-Zeiger noch unbelegt ist. Wenn ja, fügen wir einfach ein und erhöhen den Zeiger entsprechend. Wenn er kleiner als 1 wird, setzen wir ihn natürlich auf die Größe des Arrays `cap`. Analog kann man sich verhalten, wenn der `back`-Zeiger größer als die Kapazität wird.

Um die Anzahl an Elementen festzustellen, müssen wir unterscheiden, ob der `front`-Zeiger vor oder hinter dem `back`-Zeiger ist. Ist der `front`-Zeiger hinter dem `back`-Zeiger (wie in der Abbildung), so ist im Array alles ab dem `front`-Zeiger also $\texttt{cap} - \texttt{front}$ plus der gesamte vordere Bereich (also `back`) viele Elemente belegt, sonst sind genau $\texttt{back} - \texttt{front}$ viele Elemente belegt.

Zuletzt müssen wir uns noch Gedanken um das dynamische Vergrößern machen: Wenn kein Element mehr frei ist, verdoppeln wir die Arraygröße und kopieren. Wenn $\texttt{front} \leq \texttt{back}$ bildet die Schlange ein zusammenhängendes Teil im Array und wir kopieren diesen an die Indizes $0 \ldots \texttt{back} - \texttt{front}$, sonst kopieren wir den Teil bis back nach vorne und den Teil ab back nach ganz hinten im Array . Wenn wir nur noch $\frac{1}{4}$ der Elemente verwenden, verkleinern wir das Array um die Hälfte und kopieren analog.

Zuletzt müssen wir uns noch überlegen, weshalb diese Methode amortisiert konstante Kosten für alle Operationen hat. Wir verwenden hierzu die Bankkonto-Methode. Wir ignorieren `size`, `first` und `last`, da diese offensichtlich konstante Zeit benötigen. Angenommen das Array habe zurzeit Kapazität M. Wir zahlen immer 5 Euro für jedes Push und Pop auf unser Konto ein.

Wenn wir das Array vergrößern oder verkleinern, so müssen seit der letzten Vergrößerung oder Verkleinerung mindestens $\frac{M}{4}$ viele Push/Pop-Operationen durchgeführt worden sein. Das heißt wir können $4 \cdot \frac{M}{4} = M$ Operationen , die wir vorher bezahlt haben benutzen, was die benötigte Anzahl von Kosten zum Kopieren in das Größere (oder Kleinere) Array ist. Implementierung:

```java
private int[] deque;
private int front = 0;
private int back = 0;

int cap = 8;

public Deque() {
    deque = new int[cap];
}

public int size() {
    if(back < front) {
        return back + (cap - front);
    } else {
        return back - front;
    }
}

private int toIndex(int index) {
    return (index+cap)%cap;
}

public void frontPush(int elem) {
    deque[front] = elem;
    adjustSize();
    front = toIndex(front-1);
}

public void endPush(int elem) {
    adjustSize();
    deque[toIndex(back+1)] = elem;
    back = toIndex(back+1);
}

public int frontPop() {
    int element = first();
    front = toIndex(front+1); adjustSize();
    return element;
}

public int endPop() {
    int element = last();
    back = toIndex(back-1); adjustSize();
    return element;
}

private void adjustSize() {
    int newcap = cap;
    if(cap - size() <= 1) {
        newcap = cap*2;

    } else if(size() < cap/4) {
        newcap = Math.max(cap/2, 8);
    }
    if(cap == newcap) {
        return;
```

```java
        }
        int[] newList = new int[newcap];
        if(back < front) {
            for(int i = 0; i < back ; i++) {
                newList[i] = deque[i];
            }
            for(int i = front; i < cap; i++) {
                newList[i + (newcap - cap)] = deque[i];
            }
            front = front + (newcap - cap);
        } else {
            for(int i = front; i <= back ; i++) {
                newList[i - front] = deque[i];
            }
            back -= front;
            front = 0;
        }
        deque = newList;
        cap = newcap;
    }

    public int last() {
        if(back == front) {
            throw new IndexOutOfBoundsException("Empty");
        }
        return deque[toIndex(back)];
    }

    public int first() {
        if(back == front) {
            throw new IndexOutOfBoundsException("Empty");
        }
        return deque[toIndex(front+1)];
    }
}
```

Aufgabe 2

(a) Falls alle Schlüssel verschieden sind, reicht es aus beim Löschen einfach eine Markierung zu setzen die beim Suchen zusätzlich abgefragt wird.

Wir präsentieren hier eine Lua-Implementierung.

Wir nehmen an die Elemente unterstützen folgende `key` Funktion:

```lua
--[[
@param    x the element to be keyed
@returns  x                          if a number
          the key of the key field   if a table
          false                      otherwise
]]--
function key(x)
  return (type(x) == 'number' and x          ) or
         (type(x) == 'table'  and key(x.key) )
end
```

```lua
--[[
@param a the array to be searched
@param i start of the search area
@param j end  of the search area
@param x the target key
@returns 1 the element of key x, nil if not found
@returns 2 the position, nil if element not found
]]--
function search(a, i, j, x)
    if j < i then
        return nil, nil
    end
    local m = i + (j - i + 1) // 2 -- middle of search area
    local k = key(a[m])
    if not k then
        error("an element was not keyable")
    end
    if     x < k then
        return search(a, i, m-1, x)
    elseif x > k then
        return search(a, m+1, j, x)
    else
        return a[m], m
    end
end
```

```lua
--[[
Takes an existing array and adds lazy delete and find functionality
the elements in the array need to be keyable and sorted.
An element is keyable if it is a number or has a keyable field 'key'
elements must not have a field 'deleted'
]]--
function dictify(a)
    -- @returns 1 the element with key x, nil if not found or deleted
    -- @returns 2 the position if the element was found
    function a.find(self, x)
        local n = #self
        local e, p = search(self, 1, n, x)
        local valid = (type(e)=='number') or
                      (type(e)=='table' and not e.deleted) or
                      nil
        return valid and e, p
    end
    function a.delete(self, x)
        local e, p = a:find(x)
        if not e then
            return
        end
        if type(e) ~= 'table' then
            self[p] = {key = e, deleted = true}
        else
            self[p].deleted = true
        end
    end
end
```

(b) Falls Schlüssel mehrfach enthalten sind und jeweils einzeln gelöscht werden sollen, so ist es wichtig sicherzustellen, dass die Suche einen bestimmten Schlüssel auswählt und löscht. Z.B. kann man binäre Suche so anpassen, dass der gültige Schlüssel gefunden wird der am weitesten rechts steht. Hierzu setzen wir unsere Suche in der rechten Hälfte fort, wenn bereits ein gültiges Element des Schlüssels gefunden wurde während wir bei bei ungültigen Elementen die Suche in der linken Hälfte fortsetzen. Wir nutzen die Invariante, dass alle gültigen Elemente eines gleichen Schlüssels links neben allen ungültigen Elementen eines Schlüssels stehen.

```lua
--[[
@param a the array to be searched
@param i start of the search area
@param j end  of the search area
@param x the target key
@returns 1 the rightmost valid element of key x, nil if not found
@returns 2 the position, nil if element not found
]]--
function search(a, i, j, x)
    if j < i then
        return nil, nil
    end
    local m = i + (j - i + 1) // 2 -- middle of search area
    local k = key(a[m])
    if not k then
        error("an element was not keyable")
    end
    if      x < k then
        return search(a, i, m-1, x)
    elseif x > k then
        return search(a, m+1, j, x)
    else
        local e = a[m]
        local valid = (type(e)=='number') or
                      (type(e)=='table' and not e.deleted)
        if valid then
            if j == i then
                return e, m
            else
                return search(a, m, j, x)
            end
        else
            return search(a, i, m-1, x)
        end
    end
end
```

`Dictify` und `key` bleiben unverändert.

Diese Implementierung benötigt für Suchen und Löschen eine Laufzeit von $\mathcal{O}(\log N)$ wobei N die Größe des initialen Feldes ist.

Möchte man stattdessen in $\mathcal{O}(\log n)$ suchen, wobei n die aktuelle Anzahl der gültigen Schlüssel ist, so kann man wie folgt vorgehen: Man zählt die Anzahl der markierten Elemente mit. Sobald die Liste zur Hälfte markiert ist, löscht man alle markierten Elemente in einem Durchgang. Da die Anzahl der markierten Elemente zu jeder Zeit kleiner ist als die Anzahl der gültigen Elemente ist die Liste höchstens $2n$ groß, also ist die Suche in $\mathcal{O}(\log 2n) = \mathcal{O}(\log n + 1) = \mathcal{O}(\log n)$. Ebenso hat jede gewöhnliche Löschoperation eine Laufzeit von $\mathcal{O}(\log n)$. Falls eine Verkleinerung vorgenommen wird, so ist die Laufzeit $\mathcal{O}(n)$. Diese Kosten lassen sich leicht auf die restlichen Löschoperationen verteilen. Da bis zur Verkleinerung exakt n viele normale Löschoperationen stattfinden, reichen konstante Zusatzkosten pro Operation aus um das Verkleinern zu bezahlen. Folglich hat das Löschen eine amortisierte Laufzeit von $\mathcal{O}(\log n)$.

```lua
function dictify(a)
    if not a.num_deleted then a.num_deleted = 0 end
    function a.find(self, x)
        local n = #self
        local e, p = search(self, 1, n, x)
        local valid = (type(e)=='number') or
                      (type(e)=='table' and not e.deleted) or
                      nil
        return valid and e, p
    end
    function a.delete(self, x)
        local e, p = a:find(x)
        if not e then
            return
        end
        if type(e) ~= 'table' then
            self[p] = {key = e, deleted = true}
        else
            if e.deleted then
                return
            end
            self[p].deleted = true
        end
        self.num_deleted = self.num_deleted + 1
        local n = #self
        if 2 * self.num_deleted > n then
            -- cleanup
            local j = 1
            for i = 1, n do
                e = self[i]
                if (type(e)=='number') or
                   (type(e)=='table' and not e.deleted) then
                    self[j] = self[i]
                    j = j + 1
                end
            end
            while j <= n do
                self[j] = nil
                j = j + 1
            end
            self.num_deleted = 0
        end
    end
end
```

Aufgabe 3

Das Einfügen ändert sich nicht.

Das Löschen nach Schlüssel besteht einfach aus einer Suche und Löschen mit Referenz.

Zum Löschen nach Referenz verwenden wir erneut eine Strategie des Markierens. Elemente werden einfach markiert, aber wenn die Hälfte der Elemente eines Feldes markiert ist entfernen wir die markierten Elemente. Das neue Feld der halben Größe wird dann gegebenenfalls mit dem bereits vorhandenen Feld der gleichen Größe gemerged. Hat das Feld Größe 1 so wird direkt gelöscht.

Das Suchen ändert sich nur in der Form, dass die Markierung überprüft wird (Duplikate kann man dabei wie in Aufgabe 2 umgehen).

Sei N die Anzahl aller Elemente in der Datenstruktur und n die Anzahl der gültigen Elemente. Es gilt $n \leq N < 2n$.

Betrachtet man das Einfügen so verhalten sich die markierten Elemente genau so, als ob sie nicht markiert worden wären, sie behalten ihr Guthaben um für das Mergen zu bezahlen.

Nach einem Kopiervorgang beim Löschen müssen, um das Mergen bezahlen zu können, die Elemente in der neuen Liste das passende Guthaben aufweisen. Seien \$1 das Guthaben das ein Element pro Merge benötigt. Da die Elemente aus einer Liste der doppelten Größe stammen tragen sie \$1 zu wenig. Das lässt sich dadurch ausgleichen, dass die k markierten Elemente ein Guthaben von \$1 besitzen und dieses auf die k gültigen Elemente übertragen. Um dieses Guthaben aufzubauen muss jede Löschoperation \$1 bezahlen.

Entsprechend hat das Einfügen amortisierze Kosten von $\mathcal{O}(\log N) = \mathcal{O}(\log n)$.

Die Suche hat wie zuvor eine Laufzeit von $\mathcal{O}(\log^2 N) = \mathcal{O}(\log^2 n)$.

Das Löschen nach Referenz hat wenn nur markiert wird eine konstante Laufzeit. Sei $2k$ die Länge des Feldes dann hat das Löschen der markierten Elemente und das Bilden des neuen Feldes eine Laufzeit von $\mathcal{O}(k)$. Im worst case ist $\mathcal{O}(k) = \mathcal{O}(n)$. Seien 1€ die maximalen Kosten für das Überprüfen der Markierung eines Elementes und für das Kopieren eines Elementes, dann kostet das Erzeugen des neuen Feldes $3k$€. Diese Kosten lassen sich dadurch abdecken, dass jedes markierte Element 3€ an Guthaben besitzt. Um dieses Guthaben aufzubauen muss jede Löschoperation 3€ bezahlen.

Jede Löschoperation hat also nur konstant viele zusätzliche Kosten (3€ + \$1) und damit eine konstante amortisierte Laufzeit.

Für das Löschen nach Schlüssel kommt in jedem Fall eine Laufzeit von $\mathcal{O}(\log^2 n)$ für das Suchen hinzu. Die worst case Laufzeit beträgt also $\mathcal{O}(n)$ während die amortisierte Laufzeit $\mathcal{O}(\log^2 n)$ beträgt.