Arrayliste

Emilio Pielsticker

Dynamische Arrayliste I

Die Elemente einer **Arrayliste** (hier für `int`s) werden aufgeteilt auf mehrere Listenabschnitte → Datenstruktur `Segment`:

- `int elements[SEGMENT_SIZE]`: Array für Nutzdaten
- `unsigned long int len`: Anzahl belegter Plätze im Segment
- `struct Segment *pred, *succ`: Vorheriger/nächster Abschnitt

Listenabschnitte sind als **doppelt-verkettete Liste** miteinander verbunden

Dynamische Arrayliste II

Zirkuläre Verkettung:

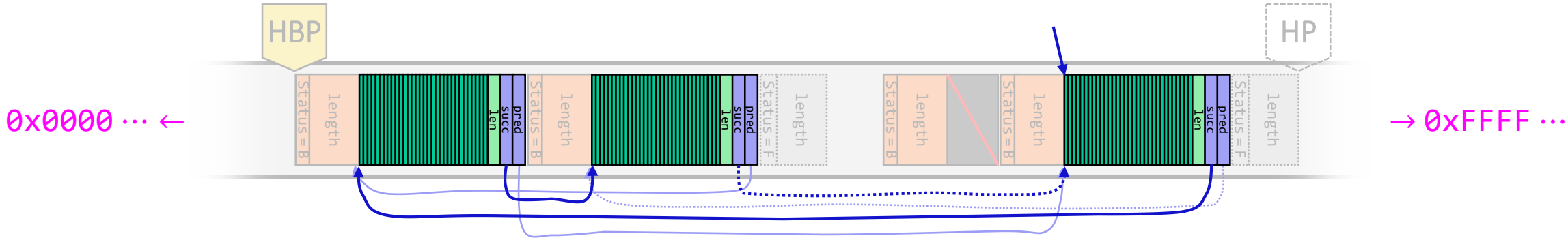
- Wenn mehr als ein Segment vorhanden ist,
  zeigt das letzte auf das erste und umgekehrt
- Bei nur einem Segment zeigen `succ` und `pred` auf NULL

Datenstruktur **Segment**

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

#define CHK(PTR, ERR, EX) ({if (!PTR) {fprintf(stderr, "\033[1;31mError:\033[0m %s\n\r", ERR); exit(EX);}})
#define CHK1(PTR) CHK(PTR, "Memory allocation failed", 125)
#define RNGERR() ({fprintf(stderr, "\033[1;31mError:\033[0m %s\n\r", "Invalid range"); exit(1);})

#define SEGMENT_SIZE 32
```

```
typedef struct Segment {
    int elements[SEGMENT_SIZE];
    unsigned long int len;
    struct Segment* pred;
    struct Segment* succ;
} segment;

typedef segment* arraylist;
```

Arrayliste im Speicher



Elemente hinzufügen

`append(list, elem)`

- Fügt `elem` ans Ende der Liste an
- Falls letzter Abschnitt voll:
  - Neuer Abschnitt wird mit `malloc()` reserviert
  - Segment wird zirkulär mit vorhandener Liste verknüpft
- Bei leerer Liste: neuer Abschnitt wird erstes Element

Elemente hinzufügen (Code)

```
void append(arraylist* list, int elem) {
    *   // if *list == NULL: Create empty list
        // Put elem to emtpy list
    segment* last = *list;
    while (last->succ && last->succ != *list) { last = last->succ; }
    *   // if last->len < SEGMENT_SIZE: Create empty segment
        // Put elem to new segment
}
```

Erzeugung neuer Listen(abschnitte) (Code)

```
// Create empty list                                    [*]
// Put elem to emtpy list

if (*list == NULL) {
    segment* seg = malloc(sizeof(segment));
    CHK1(seg);
    seg->elements[0] = elem;
    seg->len = 1;
    seg->pred = NULL;
    seg->succ = NULL;
    *list = seg;
    return;
}
```

```
if (last->len < SEGMENT_SIZE) {                    [*]
    last->elements[last->len++] = elem;
    return;
} else {
    segment* seg = malloc(sizeof(segment));
    CHK1(seg);
    seg->elements[0] = elem;
    seg->len = 1;
    seg->pred = last;
    seg->succ = *list;
    last->succ = seg;
    (*list)->pred = seg;
    return;
}
```

Letztes Element anschauen

`peek(list)`

- Gibt letztes Element der Liste zurück, ohne es zu entfernen
- Mit `last->len != 0` wird geprüft, ob das letzte Listensegment leer ist:
  Solche leeren Listensegmente sollten nicht vorkommen
  → In diesem Fall ist die Liste fehlerhaft!

Letztes Element anschauen (Code)

```
int peek(arraylist list) {
    CHK(list, "Cannot peek from empty list", 1);
    segment* last = list;
    while (last->succ && last->succ != list) { last = last->succ; }

    CHK(last->len != 0, "Last segment is empty", 1);
    return last->elements[last->len - 1];
}
```

Letztes Element entfernen

`pop(list)`

- Entfernt das letzte Element aus der Liste
- Verringert `len` im letzten Abschnitt
- Gibt Element zurück
- Falls Segment danach leer:
  - Bei letztem Segment → `free()` und Liste wird `NULL`
  - Sonst → Abschnitt entfernen, Zeiger neu setzen

Letztes Element entfernen (Code)

```
int pop(arraylist list) {
    CHK(list, "Cannot pop from empty list", 1);
    segment* last = list;
    while (last->succ && last->succ != list) { last = last->succ; }

    int value = last->elements[last->len - 1];
    last->len--;

    ⋮
}
```

Letztes Element entfernen (Fortsetzung)

```
int pop(arraylist list) {
    ⋮
 if (last->len == 0) {
        if (last == list && last->succ == NULL) {
            free(last); list = NULL;
        } else {
            last->pred->succ = last->succ;
            last->succ->pred = last->pred;
            if (last == list) { list = last->succ; }
            free(last);
        }
    }

    return value;
}
```

n-tes Element anschauen

`get(list,` n`)`

- Gibt das n-te Element zurück (n beginnt bei 0)
- Traversiert die Liste bis zum passenden Segment
- Fehler, wenn n außerhalb des gültigen Bereichs

n-tes Element anschauen (Code)

```
int get(arraylist list, unsigned long int n) {
    while (list) {
        if (n < list->len) { return list->elements[n]; }
        n -= list->len;
        list = list->succ;
        if (list && list->succ == list) { break; }
    }

    RNGERR();
}
```

n-tes Element setzen

`set(list, `n`, val)`

- Ändert das n-te Element auf Wert von `val`
- Analog zu `get()` mit direktem Schreibzugriff
- Fehler bei ungültigem Index

n-tes Element setzen (Code)

```
void set(arraylist list, unsigned long int n, int val) {
    while (list) {
        if (n < list->len) { list->elements[n] = val; return; }
        n -= list->len;
        list = list->succ;
        if (list && list->succ == list) { break; }
    }

    RNGERR();
}
```

Länge der Liste messen

`length(list)` bzw. `len(list)`

- Gibt die Gesamtanzahl an gespeicherten Elementen in der Liste zurück
- Addiert `len` jedes Segments bei Traversierung:

```
do {
      total += current->len;
      current = current->succ;
} while (current && current != list);
```

Länge der Liste messen (Code)

```
unsigned long int length(arraylist list) {
    if (!list) { return 0; }

    unsigned long int total = 0;
    segment* current = list;

    do { total += current->len;
         current = current->succ;
    } while (current && current != list);

    return total;
}

unsigned long int len(arraylist l) { return length(l); }
```

Liste in der Konsole ausgeben

`print(list)`

- Gibt gesamten Inhalt der Liste im Format `[…, …]` in der Konsole aus
- Im Falle einer leeren Liste soll `[ ]` ausgegeben werden

Liste in der Konsole ausgeben (Code)

```
void printlist(arraylist list) {
    if (!list) { printf("[]\n"); return; } else { printf("["); }
    unsigned long int first = 1;
    segment* cur = list;

    do {
        for (unsigned long i = 0; i < cur->len; i++) {
            if (!first) { printf(", "); }
            printf("%d", cur->elements[i]);
            first = 0;
        }
        cur = cur->succ;
    } while (cur && cur != list);
    printf("]\n\r")
}
```

Liste kopieren

```
void copylist(arraylist list, arraylist* dest,
              unsigned long int n, unsigned long int m) {
    CHK(n > m || len(list) <= m, "Invalid range", 1);
    for (unsigned long i = n; i<=m; i++) { append(dest, get(list, i)); }
}
```

- Kopieren einer vollständigen Liste:

```
arraylist copy = mylist;
copylist(mylist, &copy, 0, len(mylist));
```

- **Achtung**: Die hier gezeigte Kopierfunktion ist recht ineffizient…

Liste kopieren (Effizientere Variante)

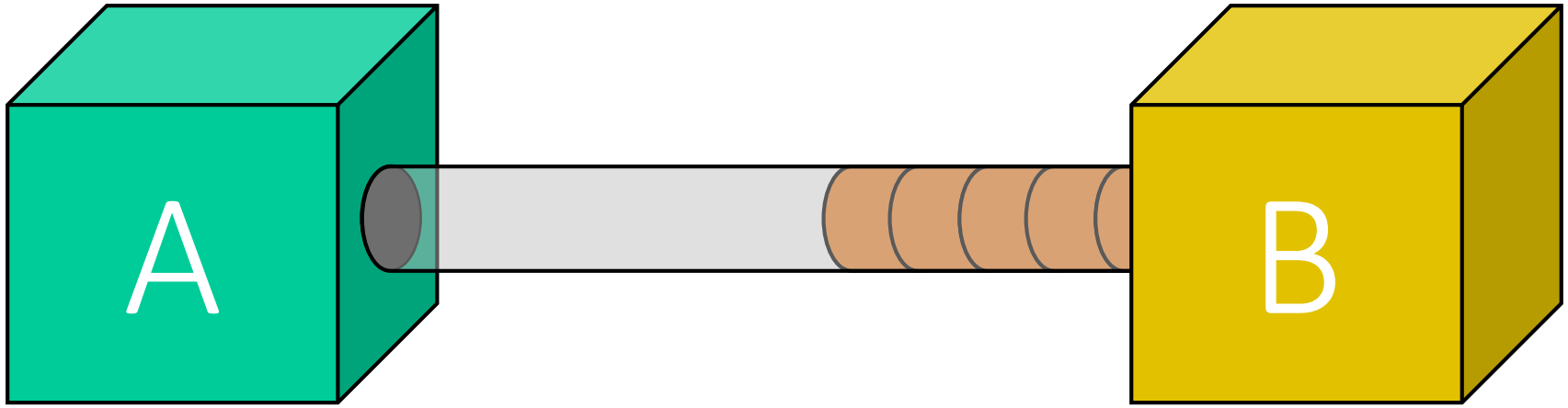
```
void copylist(arraylist list, arraylist* dest,
              unsigned long int n, unsigned long int m) {
    CHK(n > m || len(list) <= m, "Invalid range", 1);
    unsigned long index = 0;
    segment* cur = list;

    do {
        for (unsigned long i = 0; i < cur->len; i++) {
            if (index>=n && index<=m) { append(dest, cur->elements[i]); }
            if (index > m) return;
            index++;
        }
        cur = cur->succ;
    } while (cur && cur != list);
}
```

Gesamte Liste löschen

```
void del(arraylist* list) {
    if (!*list) { return; }

    segment* cur = *list;
    segment* next;
    do { next = cur->succ;
         free(cur);
         cur = next;
    } while (cur && cur != *list);

    *list = NULL;
    return;
}
```

- **Achtung**: `*list = NULL` reicht hier nicht aus! Der durch die Liste belegte Speicherplatz bleibt belegt
- Würde man über einen sog. **Garbage-Collector** (dt. **Müllsammler**) verfügen, würde dieser erkennen, dass die Liste ‚losgelassen' wurde und den Speicherplatz dann entsprechend freigeben
- In C erfolgt Heapverwaltung manuell

Feststehende Programmverbindung



Feststehende Programmverbindung (Quelle)

```
int main(int argc, char* argv[]) {
    int fd;
    srand(time(NULL));

    fd = mkfifo(FIFO_NAME, RWRWRW);
    CHK(fd, "Can't create named pipe.", 1);

    sleep(3);

    fd = open(FIFO_NAME, O_WRONLY);
    CHK(fd, "Can't open named pipe.", 1);

    :
}
```

Feststehende Programmverbindung (Quelle)

```
int main(int argc, char* argv[]) {
    ⋮

    for (int i = 0; i < N; i++) {
        int num = rand() % (RANGE + 1);
        CHK(write(fd, &num, sizeof(int)) == sizeof(int),
            "Can't write to pipe.", 1);
        sleep(rand() % 3 + 1);
    }

    close(fd);

    return 0;
}
```

Feststehende Programmverbindung (Senke)

```
int main(int argc, char* argv[]) {
    int fd;
    int number;
    arraylist mylist = NULL;

    fd = open(FIFO_NAME, O_WRONLY);
    CHK(fd, "Can't open named pipe.", 1);

    :

    close(fd);
    del(&mylist);
    return 0;
}
```

Feststehende Programmverbindung (Senke)

```
⋮
    for (int i = 0; i < N; i++) {
        ssize_t r = read(fd, &number, sizeof(int));
        CHK(r != -1, "Can't read from pipe.", 1);
        CHK(r, "Unexpected EOF", 1);

        if (number == 0) {
            if (len(mylist) > 0) { pop(&mylist); }
        } else {
            append(&mylist, number);
        }

        printlist(mylist);
    }
⋮
```