Länkade listor

Nipun Thianwan

VT 2025

1 Introduction

Den här rapporten undersöker implementationen av länkade listor i C och jämför deras prestanda med traditionella arrayer. Vi analyserar också hur man implementerar stackar med länkade listor och diskuterar för- och nackdelar. Målet är att förstå den dynamiska minneshanteringen som länkade listor kräver och analysera deras tidskomplexitet för operationer som att lägga till, ta bort och sammanfoga listor.

2 En länkad lista

I C implementeras en länkad lista genom att använda struct och pekare. En nod i listan definieras enligt följande:

```
typedef struct cell {
    int value;
    struct cell *tail;
} cell;

typedef struct linked {
    cell *first;
} linked;
```

Sen vi kan skapa en ny lista och allokera minne dynamiskt:

```
linked *linked_create() {
    linked *new = (linked*)malloc(sizeof(linked));
    new->first = NULL;
    return new;
}
```

För att lägga till ett nytt element i början av listan:

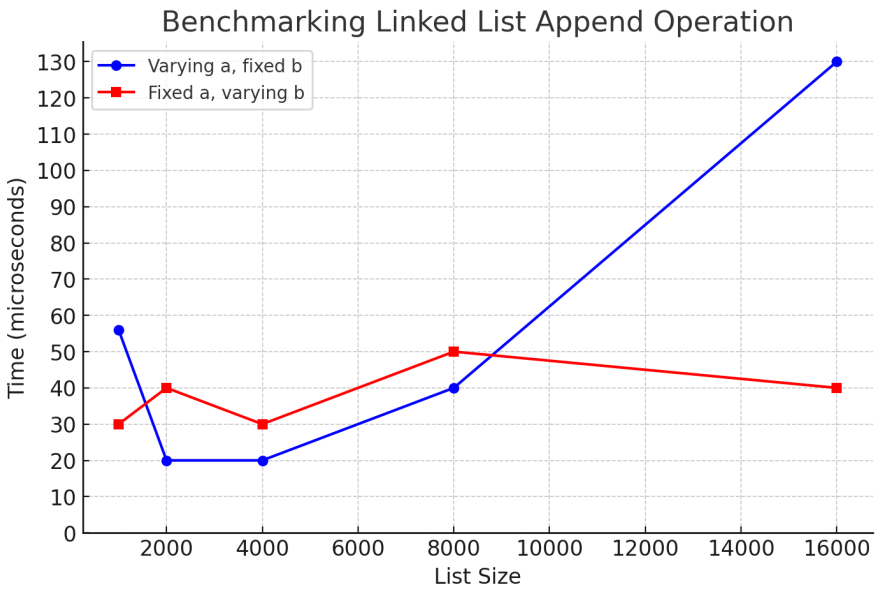
```
void linked_add(linked *lnk, int item) {
    cell *new = (cell*)malloc(sizeof(cell));
    new->value = item;
    new->tail = lnk->first;
```

```
    lnk->first = new;
}
```

och sen:

```
void linked_append(linked *a, linked *b) {
    if (a->first == NULL) {
        a->first = b->first;
    } else {
        cell *nxt = a->first;
        while (nxt->tail != NULL) {
            nxt = nxt->tail;
        }
        nxt->tail = b->first;
    }
    b->first = NULL;
}
```

2.1 benchmark



Figur 1: A varierad B fast och B varierad A fast

När vi kör denna benchmark ökar vi storleken på den första listan (a), medan den andra listan (b) har en fast storlek. Vi ser att tiden ökar linjärt, vilket tyder på att append-operationen har en tidskomplexitet av $\mathcal{O}(n)$, där n är storleken på a. Detta beror på att vi måste traversera hela a för att hitta den sista noden innan vi kan länka in b.

I nästa experiment gör vi tvärtom: vi håller a konstant och ökar storleken på b. Vi ser då att tiden förblir nästan konstant. Detta bekräftar att själva append-operationen utförs i $\mathcal{O}(1)$, eftersom vi endast uppdaterar en pekare i slutet av a.

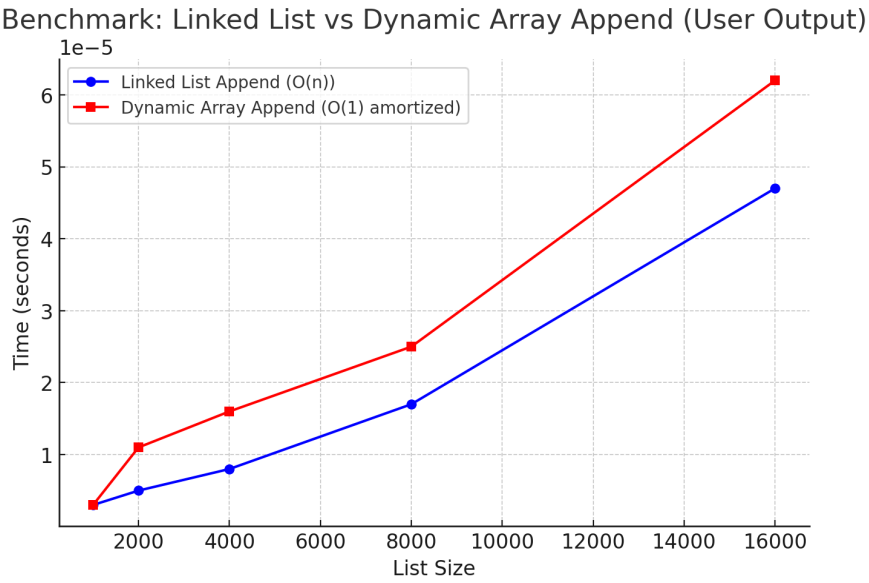
Sammanfattningsvis kan vi dra slutsatsen att den dominerande tidskomplexiteten beror på storleken av a, inte b. Om vi ville förbättra detta skulle vi kunna använda en *dubbellänkad lista* eller lagra en pekare till den sista noden, vilket skulle reducera append-operationen till $\mathcal{O}(1)$ även när a ökar.

3 Jämfört med en array

Här ska samma sak göras, men med arrayer istället. Författaren valde att implementera koden och analysera resultaten.

```
void array_append(dynamic_array *a, dynamic_array *b) {
    while (a->size + b->size > a->capacity) {
        array_resize(a);
    }
    for (int i = 0; i < b->size; i++) {
        a->data[a->size++] = b->data[i];
    }
}
```

Resultatet av den exekverade koden: När vi jämför append-operationen mellan en länkad



Figur 2: Länkad lista kontra Dynamic Array

lista och en dynamisk array ser vi att tiden för append i en länkad lista ökar linjärt med listans storlek. Detta beror på att vi måste traversera hela listan för att hitta slutet

innan vi kan lägga till den nya delen, vilket gör att tidskomplexiteten blir O(n). För en dynamisk array är append snabb i början eftersom vi kan lägga till element direkt, men när arrayen blir full måste den allokeras om och kopieras, vilket tillfälligt ökar tiden. I genomsnitt är append i en dynamisk array amortized O(1), vilket innebär att den är mer effektiv än en länkad lista vid kontinuerliga append-operationer. Sammanfattningsvis är en dynamisk array bättre om man ofta lägger till element i slutet, medan en länkad lista är mer flexibel och bättre lämpad för insättningar i början eller mitten av listan.

4 Jämförelse med stack

I den här delen av uppgiften ska vi beskriva med ord hur vi skulle implementera en stack med en länkad lista och jämföra den med en stack implementerad med en array. Vi ska diskutera hur implementationen fungerar samt deras för- och nackdelar:

När man implementerar en stack är en länkad lista en vanlig datastruktur med både fördelar och nackdelar. Den främsta fördelen är dess dynamiska storlek, vilket innebär att den kan växa och krympa efter behov utan att vara begränsad av en förutbestämd kapacitet. Detta leder till en mer effektiv minnesanvändning, eftersom minne endast allokeras när ett nytt element läggs till. Till skillnad från en arraybaserad stack, där stora minnesblock kan behöva reserveras i förväg och där en omallokering kan bli nödvändig vid expansion, undviker en länkad lista dessa problem och hanterar minnet mer flexibelt.

Trots dessa fördelar har länkade listor även nackdelar. Varje nod kräver extra minne för att lagra en pekare till nästa nod, vilket kan öka minnesförbrukningen, särskilt vid stora datamängder. Dessutom är en länkad lista mindre cachevänlig än en array, eftersom dess noder kan vara utspridda i minnet. Detta kan leda till fler cache-missar, vilket i sin tur påverkar exekveringstiden negativt. Implementationen av en länkad lista är också mer komplex, då den kräver noggrann hantering av pekare för att undvika minnesläckor och andra fel.

När det gäller prestanda är både push- och pop-operationer i en länkad lista O(1), eftersom de endast påverkar pekaren vid stackens topp. En arraybaserad stack har också O(1) för push, så länge det finns utrymme, men om arrayen blir full måste en omallokering ske, vilket gör att operationen kan ta O(n) i vissa fall.

Sammanfattningsvis är en arraybaserad stack oftast snabbare och mer minnesvänlig vid förutsägbar användning, medan en länkad lista är mer flexibel vid dynamiska storleksändringar men kan innebära högre minnesförbrukning och långsammare exekvering på grund av sämre cacheprestanda.